

Top-k Query Processing on Encrypted Databases with Strong Security Guarantees

Xianrui Meng^{*1}, Haohan Zhu^{†1}, and George Kollios^{‡1}

¹Department of Computer Science
Boston University

Abstract

Privacy concerns in outsourced cloud databases have become more and more important recently and many efficient and scalable query processing methods over encrypted data have been proposed. However, there is very limited work on how to securely process top- k ranking queries over encrypted databases in the cloud. In this paper, we focus exactly on this problem: secure and efficient processing of top- k queries over outsourced databases. In particular, we propose the first efficient and provable secure top- k query processing construction that achieves adaptively *CQA* security. We develop an encrypted data structure called **EHL** and describe several secure sub-protocols under our security model to answer top- k queries. Furthermore, we optimize our query algorithms for both space and time efficiency. Finally, in the experiments, we empirically analyze our protocol using real world datasets and demonstrate that our construction is efficient and practical.

1 Introduction

As remote storage and cloud computing services emerge, such as Amazon’s EC2, Google AppEngine, and Microsoft’s Azure, many enterprises, organizations, and end users may outsource their data to those cloud service providers for reliable maintenance, lower cost, and better performance. In fact, a number of database systems on the cloud have been developed recently that offer high availability and flexibility at relatively low costs. However, despite these benefits, there are still a number of reasons that make many users to refrain from using these services, especially users with sensitive and valuable data. Undoubtedly, the main issue for this is related to security and privacy concerns [3]. Indeed, data owner and clients may not fully trust a public cloud since some of hackers, or the cloud’s administrators with root privilege can fully access all data for any purpose. Sometimes the cloud provider may sell its business to an untrusted company, which will have full access to the data. One approach to address these issues is to encrypt the data before outsourcing them to the cloud. For example, electronic health records (EHRs) should be encrypted before outsourcing in compliance with regulations like HIPAA¹. Encrypted data can bring an enhanced security into the Database-As-Service environment [25]. However, it also introduces significant difficulties in querying and computing over these data.

Although top- k queries are important query types in many database applications [28], to the best of our knowledge, none of the existing works handle the top- k queries securely and efficiently. Vaiyda et. al. [46] studied privacy-preserving top- k queries in which the data are vertically partitioned instead of

^{*}Email: xmeng@cs.bu.edu

[†]Email: zhu@cs.bu.edu

[‡]Email: gkollios@cs.bu.edu

¹HIPAA is the federal Health Insurance Portability and Accountability Act of 1996.

encrypting the data. Wong et. al. [48] proposed an encryption scheme for knn queries and mentioned a method of transforming their scheme to solve top- k queries, however, as shown in [50], their encryption scheme is not secure and is vulnerable to chosen plaintext attacks. Vaidya et. al. [46] also studied privacy-preserving top- k queries in which the data are vertically partitioned instead of encrypting the data.

We assume that the data owner and the clients are trusted, but not the cloud server. Therefore, the data owner encrypts each database relation R using some probabilistic encryption scheme before outsourcing it to the cloud. An authorized user specifies a query q and generates a *token* to query the server. Our objective is to allow the cloud to securely compute the top- k results based on a user-defined ranking function over R , and, more importantly, the cloud should not learn anything about R or q . Consider a real world example for a health medical database below:

Example 1.1. *An authorized doctor, Alice, wants to get the top- k results based on some ranking criteria from the encrypted electronic health record database **patients** (see Table 1). The encrypted **patients** database may contain several attributes; here we only list a few in Table 1: patient name, age, id number, trestbps², chol³, thalach⁴.*

patient name	age	id	trestbps	chol	thalach
$E(\text{Bob})$	$E(38)$	$E(121)$	$E(110)$	$E(196)$	$E(166)$
$E(\text{Celvin})$	$E(43)$	$E(222)$	$E(120)$	$E(201)$	$E(160)$
$E(\text{David})$	$E(60)$	$E(285)$	$E(100)$	$E(248)$	$E(142)$
$E(\text{Emma})$	$E(36)$	$E(956)$	$E(120)$	$E(267)$	$E(112)$
$E(\text{Flora})$	$E(43)$	$E(756)$	$E(100)$	$E(223)$	$E(127)$

Table 1: Encrypted **patients** Heart-Disease Data

One example of a top- k query (in the form of a SQL query) can be: ***SELECT * FROM patients ORDERED BY chol+thalach STOP AFTER k.*** That is, the doctor wants to get the top-2 results based the score $\text{chol} + \text{thalach}$ from all the patient records. However, since this table contains very sensitive information about the patients, the data owner first encrypts the table and then delegates it to the cloud. So, Alice requests a key from the data owner and generates a query token based on the query. Then the cloud searches and computes on the encrypted table to find out the top- k results. In this case, the top-2 results are the records of patients David and Emma.

Our protocol extends the No-Random-Access (NRA) [22] algorithm for computing top- k queries over a probabilistically encrypted relational database. Moreover, our query processing model assumes that two non-colluding semi-honest clouds, which is the model that has been showed working well (see [21, 12, 36, 7, 11]). We encrypt the database in such a way that the server can obviously execute NRA over the encrypted database without learning the underlying data. This is accomplished with the help of a secondary independent cloud server (or Crypto Cloud). However, the encrypted database resides only in the primary cloud. We adopt two efficient state-of-art secure protocols, **EncSort** [7] and **EncCompare** [11], which are the two building block we need in our top- k secure construction. We choose these two building blocks mainly because of their efficiency.

During the query processing, we propose several novel sub-routines that can securely compute the best/worst score and de-duplicate replicated data items over the encrypted database. Notice that our proposed sub-protocols can also be used as stand-alone building blocks for other applications as well. We also would like to point out that during the querying phase the computation performed by the client is very small. The client only needs to compute a simple token for the server and all of the relatively

²trestbps: resting blood pressure (in mm Hg)

³chol: serum cholestoral in mg/dl

⁴maximum heart rate achieved

heavier computations are performed by the cloud side. Moreover, we also explore the problem of top- k join queries over multiple encrypted relations.

We also design a *secure top- k join operator*, denote as \bowtie_{sec} , to securely join the tables based on *equi-join* condition. The cloud homomorphically computes the top- k join on the top of joined results and reports the encrypted top- k results. Below we summarize our main contributions:

- We propose a new practical protocol designed to answer top- k queries over encrypted relational databases.
- We propose two encrypted data structures called EHL and EHL⁺ which allow the servers to homomorphically evaluate the equality relations between two objects.
- We propose several independent sub-protocols such that the clouds can securely compute the best/worst scores and de-duplicate replicated encrypted objects with the use of another non-colluding server.
- We also extend our techniques to answer top- k join queries over multiple encrypted relations.
- The scheme is experimentally evaluated using real-world datasets and result shows that our scheme is efficient and practical.

2 Related Works and Background

The problem of processing queries over the outsourced encrypted databases is not new. The work [25] proposed executing SQL queries over encrypted data in the database-service-provider model using bucketization. Since then, a number of works have appeared on executing various queries over encrypted data. One of the relevant problem related to top- k queries is the k NN (k Nearest Neighbor) queries. Note that top- k queries should not be confused with similarity search, such as k NN queries. For the k NN queries, one is interested in retrieving the k most similar objects over the database to a query object, where the similarity between two objects is measured over some metric space, for example the L_2 metric. Many works have been proposed to specifically handle k NN queries on encrypted data, such as [48, 21, 50, 15]. A significant amount of works have been done for privacy preserving keyword search queries or boolean queries, such as [44, 17, 13]. Recent work [42] proposed a general framework for boolean queries of disjunctive normal form queries on encrypted data. In addition, many works have been proposed for range queries [43, 27, 33]. Other relevant works include privacy-preserving data mining [2, 30, 47, 35, 37].

Recent works in the cryptography community have shown that it is possible to perform arbitrary computations over encrypted data, using fully homomorphic encryption (FHE) [23], or Oblivious RAM [24]. However, the performance overheads of such constructions are very high in practice, thus they're not suitable for practical database queries. Some recent advancements in ORAM schemes [41] show promise and can be potentially used in certain environments. As mentioned, [46] is the only work that studied privacy preserving execution of top- k queries. However, their approach is mainly based on the k -anonymity privacy policies, therefore, it cannot extended to encrypted databases. Recently, differential privacy [20] has emerged as a powerful model to protect against unknown adversaries with guaranteed probabilistic accuracy. However, here we consider *encrypted* data in the outsourced model; moreover, we do not want our query answer to be perturbed by noise, but we want our query result to be exact. Kuzu et. al. [32] proposed a scheme that leverages DP and leaks obfuscated access statistics to enable efficient searching. Another approach has been extensively studied is order-preserving encryption (OPE) [2, 4, 9, 40, 35], which preserves the order of the message. We note that, by definition, OPE directly reveals the order of the objects' ranks, thus does not satisfy our data privacy guarantee. Furthermore, [26] proposed a prototype for access control using deterministic proxy encryption, and other secure database systems have been proposed by using embedded secure hardware, such as TrustedDB [6] and Cipherbase [5].

3 Preliminaries

3.1 Problem Definition

Consider a data owner that has a database relation R of n objects, denoted by o_1, \dots, o_n , and each object o_i has M attributes. For simplicity, we assume that all M attributes take numerical values. Thus, the relation R is an $n \times M$ matrix. The data owner would like to outsource R to a third-party cloud S_1 that is completely untrusted. Therefore, data owner encrypts R and sends the encrypted relation ER to the cloud. After that, any authorized client should be able to get the results of the top- k query over this encrypted relation directly from S_1 , by specifying k and a score function over the M (encrypted) attributes. We consider the monotone scoring (ranking) functions that are weighted linear combinations over all attributes, that is $F_W(o) = \sum w_i \times x_i(o)$, where each $w_i \geq 0$ is a user-specified weight for the i -th attribute and $x_i(o)$ is the local score (value) of the i -th attribute for object o . Note that we consider the monotone linear function mainly because it is the most important and widely used score function on top- k queries [28]. The results of a top- k query are the objects with the highest k scores of F_W values. For example, consider an authorized client, Alice, who wants to run a top- k query over the encrypted relation ER . Consider the following query: `q = SELECT * FROM ER ORDER BY $F_W(\cdot)$ STOP AFTER k`; That is, Alice wants to get the top- k results based on her scoring function F_W , for a specified set of weights. Alice first has to request the keys from the data owner, then generates a *query token* tk . Alice sends the tk to the cloud server. The cloud server storing the encrypted database ER processes the top- k query and sends the encrypted results back to Alice. In the real world scenarios, the authorized clients can locally store the keys for generating the token.

3.2 The Architecture

We consider the secure computation on the cloud under the *semi-honest* (or *honest-but-curious*) adversarial model. Furthermore, our model assumes the existence of two *different* non-colluding semi-honest cloud providers, S_1 and S_2 , where S_1 stores the encrypted database ER and S_2 holds the secret keys and provides the crypto services. We refer to the server S_2 as the *Crypto Cloud* and assume S_2 resides in the cloud environment and is isolated from S_1 . The two parties S_1 and S_2 do not trust each other, and therefore, they have to execute secure computations on encrypted data. The two parties S_1 and S_2 belong to two different cloud providers and do not trust each other; therefore, they have to execute secure computations on encrypted data. In fact, crypto clouds have been built and used in some industrial applications today (e.g., the pCloud Crypto⁵ or boxcryptor⁶). This model is not new and has already been widely used in related work, such as [21, 12, 36, 7, 11]. As pointed out by these works, we emphasize that these cloud services are typically provided by some large companies, such as Amazon, Microsoft Azure, and Google, who have also commercial interests not to collude. The Crypto Cloud S_2 is equipped with a cryptographic processor, which stores the decryption key. The intuition behind such an assumption is as follows. Most of the cloud service providers in the market are well-established IT companies, such as Amazon AWS, Microsoft Azure and Google Cloud. Therefore, a collusion between them is highly unlikely as it will damage their reputation which effects their revenues. When the server S_1 receives the query token, S_1 initiates the secure computation protocol with the Crypto Cloud S_2 . Figure 1 shows an overview of the architecture.

3.3 Cryptographic Tools

In Table 2 we summarize the notation. In the following, we present the cryptographic primitives used in our construction.

⁵<https://www.pcloud.com/encrypted-cloud-storage.html>

⁶<https://www.boxcryptor.com/en/provider>

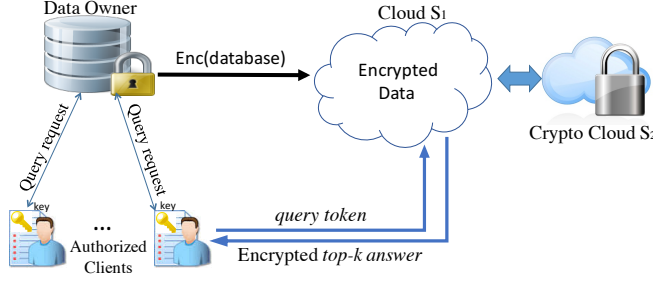


Figure 1: An overview of our model

Paillier Cryptosystem The Paillier cryptosystem [38] is a semantically secure public key encryption scheme. The message space \mathcal{M} for the encryption is \mathbb{Z}_N , where N is a product of two large prime numbers p and q . For a message $m \in \mathbb{Z}_N$, we denote $\text{Enc}_{\text{pk}}(m) \in \mathbb{Z}_{N^2}$ to be the encryption of m with the public key pk . When the key is clear in the text, we simply use $\text{Enc}(m)$ to denote the encryption of m and $\text{Dec}_{\text{sk}}(c)$ to denote the decryption of a ciphertext c . The details of encryption and decryption algorithm can be found in [38]. It has the following homomorphic properties:

- *Addition*: $\forall x, y \in \mathbb{Z}_N, \text{Enc}(x) \cdot \text{Enc}(y) = \text{Enc}(x + y)$
- *Scalar Multiplication*: $\forall x, a \in \mathbb{Z}_N, \text{Enc}(x)^a = \text{Enc}(a \cdot x)$

Generalized Paillier Our construction also relies on Damgård-Jurik(DJ) cryptosystem introduced by Damgård and Jurik [18], which is a generalization of Paillier encryption. The message space \mathcal{M} expands to \mathbb{Z}_{N^s} for $s \geq 1$, and the ciphertext space is under the group $\mathbb{Z}_{N^{s+1}}$. As mentioned in [1], this generalization allows one to doubly encrypt messages and use the additive homomorphism of the inner encryption layer under the same secret key. In particular, let $\mathcal{E}^2(x)$ denote an encryption of the DJ scheme for a message $x \in \mathbb{Z}_{N^2}$ (when $s = 2$) and $\text{Enc}(x)$ be a normal Paillier encryption. This extension allows a ciphertext of the first layer to be treated as a plaintext in the second layer. Moreover, this nested encryption preserves the structure over inner ciphertexts and allows one to manipulate it as follows:

$$\mathcal{E}^2(\text{Enc}(m_1))^{\text{Enc}(m_2)} = \mathcal{E}^2(\text{Enc}(m_1) \cdot \text{Enc}(m_2)) = \mathcal{E}^2(\text{Enc}(m_1 + m_2))$$

We note that this is the only homomorphic property that our construction relies on.

Throughout this paper, we use \sim to denote that the underlying plaintext under encryption \mathcal{E} are the same, i.e., $\text{Enc}(x) \sim \text{Enc}(y) \Rightarrow x = y$. We summarize the notation throughout this paper in Table 2. Note that in our application, we need one layered encryption; that is, given $\mathcal{E}^2(\text{Enc}(x))$, we want a normal Paillier encryption $\text{Enc}(x)$. As introduced in [7], this could simply be done with the help of S_2 . However, we need a protocol `RecoverEnc` to securely remove one layer of encryption.

3.4 No-Random-Access (NRA) Algorithm

The NRA algorithm [22] finds the top- k answers by exploiting only sorted accesses to the relation R . The input to the NRA algorithm is a set of sorted lists S , each ranks the “same” set of objects based on different attributes. The output is a ranked list of these objects ordered on the aggregate input scores. We opted to use this algorithm because it provides a scheme that leaks minimal information to the cloud server (since during query processing there is no need to access intermediate objects). We assume that each column (attribute) is sorted independently to create a set of sorted lists S . The set of sorted lists is equivalent to the original relation, but the objects in each list L are sorted in ascending order according to their local score (attribute value). After sorting, R contains M sorted lists, denoted as $S = \{L_1, L_2, \dots, L_M\}$. Each sorted list consists of n data items, denoted as $L_i = \{I_i^1, I_i^2, \dots, I_i^n\}$. Each data item is a object/value pair $I_i^d = (o_i^d, x_i^d)$, where o_i^d and x_i^d are the object id and local score at

Notation	Definition
n	Size of the relation R , i.e. $ R = n$
M	Total number of attributes in R
m	Total number of attributes for the query q
$\text{Enc}(m)$	Paillier encryption of m
$\text{Dec}(c)$	Paillier decryption of c
$\mathcal{E}^2(m)$	Damgård-Jurik (DJ) encryption of m
$\text{Enc}(x) \sim \text{Enc}(y)$	Denotes $x = y$, i.e. $\text{Dec}(\text{Enc}(x)) = \text{Dec}(\text{Enc}(y))$
$\text{EHL}(o)$	Encrypted Hash List of the object o
$\text{EHL}^+(o)$	Efficient Encrypted Hash List of the object o
\ominus, \odot	EHL and EHL^+ operations, see Section 5.
I_i^d	The data item in the i th sorted list L_i at depth d
$E(I_i^d)$	Encrypted data item I_i^d
$F_W(o)$	Cost function in the query token
$B^d(o)$	The best score (upper bound) of o at depth d
$W^d(o)$	The worst score (lower bound) of o at depth d

Table 2: Notation Summarization

Algorithm 1: NRA Algorithm [22]

```

1 def  $NRA(L_1, \dots, L_M)$ :
2   Do sorted access in parallel to each of the  $M$  sorted lists  $L_i$ . At each depth  $d$ : repeat
3     Maintain the bottom values  $\underline{x}_1^d, \underline{x}_2^d, \dots, \underline{x}_M^d$  encountered in the lists;
4     For every object  $o_i$  compute a lower bound  $W^d(o_i)$  and upper bound  $B^d(o_i)$ ;
5     Let  $T_k^d$ , the current top  $k$  list, contain the  $k$  objects with the largest  $W^d(\cdot)$  values seen so far (and their grades), and let  $M_k^d$  be the  $k$ th largest lower bound value,  $W^d(\cdot)$  in  $T_k^d$ ;
6     Halt and return  $T_k^d$  when at least  $k$  distinct objects have been seen (so that in particular  $T_k^d$  contains  $k$  objects) and when  $B^d(o_k) \leq M_k^d$  for all  $o_k \notin T_k^d$ , i.e the upper bound for every object who's not in  $T_k^d$  is no greater than  $M_k^d$ . Otherwise, go to next depth;
7   until;
```

the depth d (when d objects have been accessed under sorted access in each list) in the i th sorted list respectively. Since it produces the top- k answers using bounds computed over their exact scores, NRA may not report the exact object scores. The score lower bound of some object o , $W(o)$, is obtained by applying the ranking function on o 's known scores and the minimum possible values of o 's unknown scores. The score upper bound of o , $B(o)$, is obtained by applying the ranking function on o 's known scores and the maximum possible values of o 's unknown scores, which are the same as the last seen scores in the corresponding ranked lists. The algorithm reports a top- k object even if its score is not precisely known. Specifically, if the score lower bound of an object o is not below the score upper bounds of all other objects (including unseen objects), then o can be safely reported as the next top- k object. We give the details of the NRA in Algorithm 1.

4 Scheme Overview

In this section, we give an overview of our scheme. The two non-colluding semi-honest cloud servers are denoted by S_1 and S_2 .

Definition 4.1. Let $\text{SecTopK} = (\text{Enc}, \text{Token}, \text{SecQuery})$ be the secure top- k query scheme containing

three algorithms **Enc**, **Token** and **SecQuery**.

- **Enc**(λ, R): is the probabilistic encryption algorithm that takes relation R and security parameter λ as its inputs and outputs the encrypted relation **ER** and secret key K .
- **Token**(K, q): takes a query q and secret key K . It outputs a token **tk** for the query q .
- **SecQuery**(**tk**, **ER**) is the query processing algorithm that takes the token **tk** and **ER** and securely computes top- k results based on the **tk**.

As mentioned earlier, our encryption scheme takes advantage of the NRA top- k algorithm. The idea of **Enc** is to encrypt and permute the set of sorted lists for R , so that the server can execute a variation of the NRA algorithm using only sequential accesses to the encrypted data. To do this encryption, we design a new encrypted data structure for the objects, called **EHL**. The **Token** computes a token that serves as a trapdoor so that the cloud knows which list to access. In **SecQuery**, S_1 scans the encrypted data depth by depth for each targeted list, maintaining a list of encrypted top- k object ids per depth until there are k encrypted object ids that satisfy the NRA halting condition. During this process, S_1 and S_2 learn nothing about the underlying scores and objects. At the end of the protocol, the object ids can be reported to the client. As we discuss next, there are two options after that. Either the encrypted records are retrieved and returned to the client, or the client retrieves the records using oblivious RAM [24] that does not even reveal the location of the actual encrypted records. In the first case, the server can get some additional information by observing the access patterns, i.e., the encrypted results of different queries. However, there are schemes that address this access leakage [29, 32] and is beyond the scope of this paper. The second approach may be more expensive but is completely secure.

In the following sections, we first discuss the new encrypted data structures **EHL** and **EHL**⁺. Then, we present the three algorithms **Enc**, **Token** and **SecQuery** in more details.

5 Encrypted Hash List (EHL)

In this paper, we propose a new data structure called encrypted hash list (**EHL**) to encrypt each object. The main purpose of this structure is to allow the cloud to homomorphically compute equality between the objects, whereas it is computationally hard for the server to figure out what the objects are. Intuitively, the idea is that given an object o we use s Pseudo-Random Function (PRF) to hash the object into a binary list of length H and then encrypt all the bits in the list to generate **EHL**. In particular, we use the secure key-hash functions **HMAC** as the PRFs. Let **EHL**(o) be the encrypted list of an object o and let **EHL**(o)[i] denote the i th encryption in the list. In particular, we initialize an empty list **EHL** of length H and fill all the entries with 0. First, we generate s secure keys $\kappa_1, \dots, \kappa_s$. The object o is hashed to a list as follows: 1) Set **EHL**[**HMAC**(κ_i, o) mod H] = 1 for $1 \leq i \leq s$. 2) Encrypt each bit using Paillier encryption: for $0 \leq j \leq H - 1$, **Enc**(**EHL**(o)[j]). Fig. 2 shows how we obtain **EHL**(o) for the object o .

Lemma 5.1. *Given two objects o_1 and o_2 , their **EHL**(o_1) and **EHL**(o_2) are computationally indistinguishable.*

It is obvious to see that Lemma 5.1 holds since the bits in the **EHL** are encrypted by the semantically secure Paillier encryption scheme. Given **EHL**(x) and **EHL**(y), we define the *randomized operation* \ominus between **EHL**(x) and **EHL**(y) as follows:

$$\mathbf{EHL}(x) \ominus \mathbf{EHL}(y) \stackrel{\text{def}}{=} \prod_{i=0}^{H-1} (\mathbf{EHL}(x)[i] \cdot \mathbf{EHL}(y)[i]^{-1})^{r_i} \quad (1)$$

where each r_i is some random value in \mathbb{Z}_N .

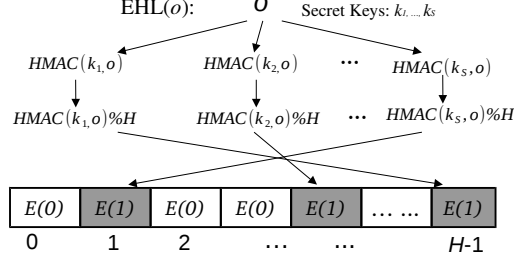


Figure 2: Encrypted Hash List for the object o .

Lemma 5.2. *Let $\text{Enc}(b) = \text{EHL}(x) \ominus \text{EHL}(y)$. Then the plaintext $b = 0$ if $x = y$ (two objects are the same), otherwise b is uniformly distributed in the group \mathbb{Z}_N with high probability.*

Proof: Let $\text{Enc}(x_i) = \text{EHL}(x)[i]$ and $\text{Enc}(y_i) = \text{EHL}(y)[i]$. If $x = y$, i.e. they are the same objects, then for all $i \in [0, H - 1]$, $x_i = y_i$. Therefore,

$$\prod_{i=0}^{H-1} (\text{EHL}(x)[i] \cdot \text{EHL}(y)[i]^{-1})^{r_i} = \mathcal{E}\left(\sum_{i=0}^{H-1} (r_i(x_i - y_i))\right) = \text{Enc}(0)$$

In the case of $x \neq y$, it must be true, with high probability, that there exists some $i \in [0, H - 1]$ such that $\text{Enc}(x_i) \neq \text{Enc}(y_i)$, i.e. the underlying bit at location i in $\text{EHL}(x)$ is different from the bit in $\text{EHL}(y)$. Suppose $\text{EHL}(x)[i] = \text{Enc}(1)$ and $\text{EHL}(y)[i] = \text{Enc}(0)$. Therefore, the following holds:

$$(\text{EHL}(x)[i] \cdot \text{EHL}(y)[i]^{-1})^{r_i} = \text{Enc}(r_i(1 - 0)) = \text{Enc}(r_i)$$

Hence, based on the definition \ominus , it follows that b becomes random value uniformly distributed in the group \mathbb{Z}_N . \blacksquare

It is worth noting that one can also use BGN cryptosystem for the similar operations above, as the BGN scheme can homomorphically evaluate quadratic functions.

False Positive Rate. Note that the construction is indeed a probabilistically encrypted Bloom Filter except that we use one list for each object and encrypt each bit in the list. The construction of EHL may report some false positive results for its \ominus operation, i.e. $\text{Enc}(0) \leftarrow \text{EHL}(x) \ominus \text{EHL}(y)$ when $x \neq y$. This is due to the fact that x and y may be hashed to exactly the same locations using s many HMACs. Therefore, it is easy to see that the false positive rate (FPR) is the same as the FPR of the Bloom Filter, where we can choose the number of hash functions HMAC s to be $\frac{H}{n} \ln 2$ to minimize the false positive rate to be $(1 - (1 - \frac{1}{H})^{sn})^s \approx (1 - e^{-sn/H})^s \approx 0.62^{H/n}$. To reduce the false positive rate, we can increase the length of the list H . However, this will increase the cost of the structure both in terms of space overhead and number of operations for the randomization operation which is $O(H)$. In the next subsection, we introduce a more compact and space-efficient encrypted data structure EHL^+ .

EHL⁺. We now present a computation- and space-efficient encrypted hash list EHL^+ . The idea of the efficient EHL^+ is to first ‘securely hash’ the object o to a larger space s times and only encrypt those hash values. Therefore, for the operation \ominus , we only homomorphically subtract those hashed values. The complexity now reduces to $O(s)$ as opposed to $O(H)$, where s is the number of the secure hash functions used. We show that one can get negligible false positive rate even using a very small s . To create an $\text{EHL}^+(o)$ for an object o , we first generate s secure keys k_1, \dots, k_s , then initialize a list EHL^+ of size s . We first compute $o_i \leftarrow \text{HMAC}(k_i, o) \bmod N$ for $1 \leq i \leq s$. This step maps o to an element in the group \mathbb{Z}_N , i.e. the message space for Paillier encryption. Then set $\text{EHL}^+[i] \leftarrow \text{Enc}(o_i)$

for $1 \leq i \leq s$. The operation \ominus between $\text{EHL}^+(x)$ and $\text{EHL}^+(y)$ are similar defined as in Equation(1), i.e. $\text{EHL}^+(x) \ominus \text{EHL}^+(y) \stackrel{\text{def}}{=} \prod_{i=0}^{s-1} (\text{EHL}(x)[i] \cdot \text{EHL}(y)[i]^{-1})^{r_i}$, where each r_i is some randomly generated value in \mathbb{Z}_N . Similarly, EHL^+ has the same properties as EHL . Let $\text{Enc}(b) \leftarrow \text{EHL}^+(x) \ominus \text{EHL}^+(y)$, $b = 0$ if $x = y$ and otherwise b is random in \mathbb{Z}_N with high probability.

We now analyze the false positive rate (FPR) for EHL^+ . The false positive answer occurs when $x \neq y$ and $\text{Enc}(0) \leftarrow \text{EHL}^+(x) \ominus \text{EHL}^+(y)$. That is $\text{HMAC}(k_i, x) \% N = \text{HMAC}(k_i, y) \% N$ for each $i \in [1, s]$. Assuming HMAC is a Pseudo-Random Function, the probability of this happens is at most $\frac{1}{N^s}$. Taking the union bound gives that the FPR is at most $\binom{n}{2} \frac{1}{N^s} \leq \frac{n^2}{N^s}$. Notice that $N \approx 2^\lambda$ is large number as N is the product of two large primes p and q in the Paillier encryption and λ is the security parameter. For instance, if we set N to be a 256 bit number (128-bit primes in Paillier) and set $s = 4$ or 5, then the FPR is negligible even for millions of records. In addition, the size of the EHL^+ is much smaller than EHL as it stores only s encryptions. In the following section, we simply say EHL to denote the encrypted hash list using the EHL^+ structure.

Notation. We introduce some notation that we use in our construction. Let $\mathbf{x} = (x_1, \dots, x_s) \in \mathbb{Z}_N^s$ and let the encryption $\text{Enc}(\mathbf{x})$ denotes the concatenation of the encryptions $\text{Enc}(x_1) \dots \text{Enc}(x_s)$. Also, we denote by \odot the block-wise multiplication between $\text{Enc}(\mathbf{x})$ and $\text{EHL}(y)$; that is, $\mathbf{c} \leftarrow \text{Enc}(\mathbf{x}) \odot \text{EHL}(y)$, where $\mathbf{c}_i \leftarrow \text{Enc}(x_i) \cdot \text{EHL}(y)[i]$ for $i \in [1, s]$.

6 Database Encryption

We describe the database encryption procedure Enc in this section. Given a relation R with M attributes, the data owner first encrypts the relation using Algorithm 2.

Algorithm 2: $\text{Enc}(R)$: Relation encryption

- 1 Given the relation R , sort each L_i based on the attribute's value for $1 \leq i \leq M$;
 - 2 Generate a public/secret key pk_p, sk_p for the Paillier encryption scheme and random secret keys $\kappa_1, \dots, \kappa_s$ for EHL ;
 - 3 Do sorted access in parallel to each of the M sorted lists L_i ;
 - 4 **foreach** data item $I_i = \langle o_i^d, x_i^d \rangle \in L_i$ **do**
 - 5 **foreach** depth d **do**
 - 6 Compute $\text{EHL}(o_i^d)$ using the keys $\kappa_1, \dots, \kappa_s$;
 - 7 Compute $\text{Enc}_{\text{pk}_p}(x_i^d)$ using pk_p ;
 - 8 Store the item $E(I_i^d) = \langle \text{EHL}(o_i^d), \text{Enc}_{\text{pk}_p}(x_i^d) \rangle$ at depth d ;
 - 9 Generate a secret key K for a pseudorandom permutation P and permute all the list based on g .
For $1 \leq i \leq M$, permute L_i as $L_{P_K(i)}$;
 - 10 The data owner securely uploads the keys pk_p, sk_p to the S_2 , and only pk_p to S_1 ;
 - 11 Finally, each permuted list contains a list of encrypted item of the form
 $E(I^d) = \langle \text{EHL}(o^d), \text{Enc}_{\text{pk}_p}(x^d) \rangle$. Output all lists of encrypted items as the encrypted relation as ER ;
-

In ER each data item $I_i^d = (o_i^d, x_i^d)$ at depth d in the sorted list L_i is encrypted as $E(I_i^d) = \langle \text{EHL}(o_i^d), \text{Enc}_{\text{pk}_p}(x_i^d) \rangle$. As all the score has been encrypted under the public key pk_p , for the rest of the paper, we simply use $\text{Enc}(x)$ to denote the encryption $\text{Enc}_{\text{pk}_p}(x)$ under the public key pk_p . Besides the size of the database and M , the encrypted ER doesn't reveal anything. In Theorem 6.1, we demonstrate this by showing that two encrypted databases are indistinguishable if they have the same size and number of attributes. We denote $|R|$ by the size of a relation R .

Theorem 6.1. *Given two relations R_1 and R_2 with $|R_1| = |R_2|$ and same number of attributes. The encrypted ER_1 and ER_2 output by the algorithm **Enc** are indistinguishable.*

The proof is straight forward as it's easy to see that the theorem holds based on Lemma 5.1 and Paillier encryption scheme.

7 Query Token

Consider the SQL-like query $q = \text{SELECT } * \text{ FROM } ER \text{ ORDERED BY } F_W(\cdot) \text{ STOP BY } k$, where $F_W(\cdot)$ is a weighted linear combination of all attributes. In this paper, to simplify our presentation of the protocol, we consider binary weights and therefore the scoring function is just a sum of the values of a subset of attributes. However, notice that for non $\{0, 1\}$ weights the client should provide these weights to the server and the server can simply adapt the same techniques by using the scalar multiplication property of the Paillier encryption before it performs the rest of the protocol which we discuss next. On input the key K and query q , the $\text{Token}(K, q)$ algorithm is quite simple and works as follows: the client specifies the scoring attribute set M of size m , i.e. $|M| = m \leq M$, then requests the key K from the data owner, where K is the key corresponds the Pseudo Random Permutation P . Then the client computes the $P_K(i)$ for each $i \in M$ and sends the following query token to the cloud server S_1 : $tk = \text{SELECT } * \text{ FROM } ER \text{ ORDERED BY } \{P_K(i)\}_{i \in M} \text{ STOP BY } k$.

8 Top-k Query Processing

As mentioned, our query processing protocol is based on the NRA algorithm. However, the technical difficulty is to execute the algorithm on the encrypted data while S_1 does not learn any object id or any score and attribute value of the data. We incorporate several cryptographic protocols to achieve this. Our query processing uses two state-of-the-art efficient and secure protocols: **EncSort** introduced by [7] and **EncCompare** introduced by [11] as building blocks. We skip the detailed description of these two protocols since they are not the focus of this paper. Here we only describe their functionalities: 1). **EncSort**: S_1 has a list of encrypted keyed-value pairs $(\text{Enc}(key_1), \text{Enc}(a_1)) \dots (\text{Enc}(key_m), \text{Enc}(a_m))$ and a public key pk , and S_2 has the secret key sk . At the end of the protocol, S_1 obtains a list *new* encryptions $(\text{Enc}(key'_1), \text{Enc}(a'_1)) \dots (\text{Enc}(key'_m), \text{Enc}(a'_m))$, where the key/value list is sorted based on the order $a'_1 \leq a'_2 \dots \leq a'_m$ and the set $\{(key_1, a_1), \dots, (key_m, a_m)\}$ is the same as $\{(key'_1, a'_1), \dots, (key'_m, a'_m)\}$. 2). **EncCompare**($\text{Enc}(a), \text{Enc}(b)$): S_1 has a public key pk and two encrypted values $\text{Enc}(a), \text{Enc}(b)$, while S_2 has the secret key sk . At the end of the protocol, S_1 obtains the bit f such that $f := (a \leq b)$. Several protocols have been proposed for the functionality above. We choose the one from [11] mainly because it is efficient and perfectly suits our requirements.

8.1 Query Processing: SecQuery

We first give the overall description of the top- k query processing **SecQuery** at a high level. Then in Section 8.2, we describe in details the secure sub-routines that we use in the query processing: **SecWorst**, **SecBest**, **SecDedup**, and **SecUpdate**.

As mentioned, **SecQuery** makes use of the NRA algorithm but is different from the original NRA, because **SecQuery** cannot maintain the global worst/best scores in plaintext. Instead, **SecQuery** has to run secure protocols depth by depth and homomorphically compute the worst/best scores based on the items at each depth. It then has to update the complete list of encrypted items seen so far with their global worst/best scores. At the end, server S_1 reports k encrypted objects (or object ids) without learning any object or its scores.

Notations. In the encrypted database, we denote each *encrypted item* by $E(I) = \langle \text{EHL}(o), \text{Enc}(x) \rangle$, where I is the item with object id o and score x . During the query processing, the server S_1 needs to maintain the encrypted item with its current best/worst scores, and we denote by $\mathbf{E}(I) = (\text{EHL}(o), \text{Enc}(W), \text{Enc}(B))$ the *encrypted score item* I with object id o with best score B and worst score W .

Algorithm 3: Top- k Query Processing: SecQuery

```

1  $S_1$  receives Token from the client;
2 Parses the Token and let  $L_i = L_{P_K(j)}$  for  $j \in M$ ;
3 foreach depth  $d$  at each list do
4   foreach  $E(I_i^d) = \langle \text{EHL}(o_i^d), \text{Enc}(x_i^d) \rangle \in L_i$  do
5     /* Compute the worst score for object  $o_i^d$  at current depth  $d$  */
6     Compute  $\text{Enc}(W_i^d) \leftarrow \text{SecWorst}(E(I_i^d), H, \text{pk}_p, \text{sk}_p)$ , where  $H = \{E(I_j^d)\}_{j \in M, i \neq j}$ ;
7     /* Compute the best score for object  $o_i^d$  at current depth  $d$  */
8     Compute  $\text{Enc}(B_i^d) \leftarrow \text{SecBest}(E(I_i^d), \{j\}_{j \neq i}, \text{pk}_p, \text{sk}_p)$ ;
9     /* gets encrypted list  $\Gamma_d$  without duplicated objects */
10    Run  $\Gamma_d \leftarrow \text{SecDedup}(\{\mathbf{E}(I_i^d)\}, \text{pk}_p, \text{sk}_p)$  with  $S_2$  and get the local encrypted list  $\Gamma_d$ ;
11    Run  $T^d \leftarrow \text{SecUpdate}(T^{d-1}, \Gamma_d, \text{pk}_p, \text{sk}_p)$  with  $S_2$  and get  $T^d$ ;
12    If  $|T^d| < k$  elements, go to the next depth. Otherwise, run  $\text{EncSort}(T^d)$  by sorting on
13     $\text{Enc}(W_i)$ , get first  $k$  items as  $T_k^d$ ;
14    Let the  $k$ th and the  $(k+1)$ th item be  $E(I'_k)$  and  $E(I'_{k+1})$ ,  $S_1$  then runs
15     $f \leftarrow \text{EncCompare}(E(W'_k), E(B'_{k+1}))$  with  $S_2$ , where  $E(W'_k)$  is the worst score for  $E(I'_k)$ , and
16     $E(B'_{k+1})$  is the best score for  $E(I'_{k+1})$  in  $T^d$ ;
17    if  $f = 0$  then
18      Halt and return the encrypted first  $k$  item in  $T_k^d$ 

```

In particular, upon receiving the token $\text{tk} = \text{SELECT } * \text{ FROM ER ORDERED BY } \{P_K(i)\}_{i \in M} \text{ STOP BY } k$, the cloud server S_1 begins to process the query. The token tk contains $\{P_K(i)\}_{i \in M}$ which informs S_1 to perform the sequential access to the lists $\{L_{P_K(i)}\}_{i \in M}$. By maintaining an encrypted list T , which includes items with their encrypted global best and worst scores, S_1 updates the list T depth by depth. Let T^d be the state of the encrypted list T after depth d . At depth d , S_1 first homomorphically computes the local encrypted worst/best scores for each item appearing at this depth by running SecWorst and SecBest .

In SecWorst , S_1 takes the input of the current encrypted item $E(I_i^d) = \langle \text{EHL}(o_i^d), \text{Enc}(x_i^d) \rangle$ and all of the encrypted items in other lists H at current depth, i.e., $H = \{E(I_j^d)\}_{j \neq i, j \in M}$. S_1 runs the protocol SecWorst with S_2 , and obtains the encrypted worst score for the object o_i^d . Similarly, in the protocol SecBest , S_1 takes the input of the current encrypted item $E(I_i^d) = \langle \text{EHL}(o_i^d), \text{Enc}(x_i^d) \rangle$ and the list pointers $\{j\}_{j \neq i}$ that indicates all of the encrypted item seen so far. S_1 runs the protocol SecBest with S_2 , and obtains the encrypted worst score for the object o_i^d . Then S_1 securely replaces the duplicated encrypted objects with large encrypted worst scores Z by running SecDedup with S_2 . In the SecDedup protocol, S_1 inputs the current encrypted items, $\{E(I_i^d)\}$, seen so far. After the execution of the protocol, S_1 gets list of encrypted items Γ_d such that there are no duplicated objects. Next, S_1 updates the encrypted global list from state T^{d-1} to state T^d by applying SecUpdate . After that, S_1 utilizes EncSort to sort the distinct encrypted objects with their scores in T^d to obtain the first k encrypted objects which are essentially the top- k objects based on their worst scores so far. The protocol halts if at some depth, the encrypted best score of the $(k+1)$ -th object, $\text{Enc}(B_{k+1})$, is less than the k -th object's encrypted worst score $\text{Enc}(W_k)$. This can be checked by calling the protocol $\text{EncCompare}(\text{Enc}(W_k), \text{Enc}(B_{k+1}))$. Followed by underlying NRA algorithm, it is easy to see that S_1 can

correctly reports the encrypted top- k objects. We describe the detailed query processing in Algorithm 3.

8.2 Building Blocks

In this section, we present the detailed description of the protocols SecWorst, SecBest, SecDedup, and SecUpdate.

8.2.1 Secure Worst Score

At each depth, for each encrypted data item, server S_1 should obtain the encryption $\text{Enc}(W)$, which is the worst score based on the items at the current depth *only*. Note that this is different than the normal NRA algorithm as it computes the global worst possible score for each encountered objects until the current depth. We formally describe the protocol setup below:

Protocol 8.1. *Server S_1 has the input $E(I) = \langle \text{EHL}(o), \text{Enc}(x) \rangle$, a set of encrypted items H , i.e. $H = \{E(I_i)\}_{i=1}^{|H|}$, where $E(I_i) = \langle \text{EHL}(o_i), \text{Enc}(x_i) \rangle$, and the public key pk_p . Server S_2 's inputs are pk_p and sk_p . SecWorst securely computes the encrypted worst ranking score based on L , i.e., S_1 outputs $\text{Enc}(W(o))$, where $W(o)$ is the worst score based on the list H .*

The technical challenge here is to homomorphically evaluate the encrypted score only based on the objects' equality relation. That is, if the object is the same as another o from L , then we add the score to $\text{Enc}(W(o))$, otherwise, we don't. However, we want to prevent the servers from knowing the relations between the objects at any depth. We overcome this problem using the protocol SecWorst($E(I), L$) between the two servers S_1 and S_2 . We present the detailed protocol description of SecWorst in Algorithm 4.

Algorithm 4: SecWorst($E(I), H = \{E(I_i)\}_{i=1}^{|H|}, \text{pk}_p, \text{sk}_p$): Worst Score Protocol

S_1 's input: $E(I), H = \{E(I_j)\}, \text{pk}_p$
 S_2 's input: pk_p, sk_p

1 **Server S_1 :**
2 Let $|H| = m$. Generate a random permutation $\pi : [m] \rightarrow [m]$;
3 For the set of encrypted items $H = \{E(I_j)\}$, permute each $E(I_j)$ in H as
 $E(I_{\pi(j)}) = \langle \text{EHL}(o_{\pi(j)}), \text{Enc}(x_{\pi(j)}) \rangle$;
4 **for each permuted item in $E(I_{\pi(j)})$ do**
5 compute $\text{Enc}(b_j) \leftarrow \text{EHL}(o) \ominus \text{EHL}(o_{\pi(j)})$, send $\text{Enc}(b_j)$ to S_2
6 Receive $\mathcal{E}^2(t_i)$ from S_2 and evaluate: $\mathcal{E}^2(\text{Enc}(x'_i)) := \mathcal{E}^2(t_i)^{\text{Enc}(x_i)} \cdot \left(\mathcal{E}^2(1) \mathcal{E}^2(t_i)^{-1} \right)^{\text{Enc}(0)}$;
7 Run $\text{Enc}(x'_i) \leftarrow \text{RecoverEnc}(\mathcal{E}^2(\text{Enc}(x'_i)), \text{pk}_p, \text{sk}_p)$ with S_2 ;
8 Set the worst score $\text{Enc}(W) \leftarrow (\prod_{i=1}^m \text{Enc}(x'_i))$;
9 Output $\text{Enc}(W)$.

10 **Server S_2 :**
11 **for each $\text{Enc}(b_i)$ received from S_1 do**
12 Decrypt to get b_i , set $t_i \leftarrow (b_i = 0 ? 1 : 0)$;
13 Send $\mathcal{E}^2(t_i)$ to S_1 .

Intuitively, the idea of SecWorst is that S_1 first generates a random permutation π and permutes the list of items in L . Then, it computes the $\text{Enc}(b_i)$ between $E(I)$ and each permuted $E(I_{\pi(i)})$, and sends $\text{Enc}(b_i)$ to S_2 . The random permutation prevents S_2 from knowing the pair-wise relations between o and the rest of the objects o_i 's. Then S_2 sends $\mathcal{E}^2(t_i)$ to S_1 (line 13). Based on Lemma 5.2,

$t_i = 1$ if two objects are the same, otherwise $t_i = 0$. S_1 then computes $\mathcal{E}^2(\text{Enc}(x'_i)) \leftarrow \mathcal{E}^2(t_i)^{\text{Enc}(x_i)}$. $(\mathcal{E}^2(1)\mathcal{E}^2(t_i)^{-1})^{\text{Enc}(0)}$. Based on the properties of DJ Encryption,

$$\mathcal{E}^2(t_i)^{\text{Enc}(x_i)} \cdot (\mathcal{E}^2(1)\mathcal{E}^2(t_i)^{-1})^{\text{Enc}(0)} = \mathcal{E}^2(t_i \cdot \text{Enc}(x_i) + (1 - t_i) \cdot \text{Enc}(0)) = \mathcal{E}^2(\text{Enc}(x'_i))$$

Therefore, it follows that $x'_i = 0$ if $t_i = 0$, otherwise $x'_i = x_i$. S_1 then runs $\text{RecoverEnc}(\mathcal{E}^2(\text{Enc}(x'_i)), \text{pk}_p, \text{sk}_p)$ (describe in Algorithm 5) to get $\text{Enc}(x'_i)$. Note that the protocol RecoverEnc is also used in other protocols. Finally, S_1 evaluates the following equation: $\text{Enc}(W(o)) \leftarrow \prod_{i=1}^m \text{Enc}(x'_i)$. S_1 can correctly evaluate the worst score, because that, when $t_i = 0$, the object o_i is not the same as o , otherwise, $t_i = 1$. The following formula gives the correct computation of the worst score:

$$\prod_{i=1}^m \text{Enc}(x'_i) = \text{Enc}(\sum_{i=1}^m x'_i), \text{ where } x'_i = \begin{cases} x_i & \text{if } o_i = o \\ 0 & \text{otherwise} \end{cases}$$

Algorithm 5: $\text{RecoverEnc}(\mathcal{E}^2(\text{Enc}(c)), \text{pk}_p, \text{sk}_p)$ Recover Encryption

S_1 's input: $\mathcal{E}^2(\text{Enc}(c)), \text{pk}_p$

S_2 's input: pk_p, sk_p

1 **Server S_1 :**

2 Generate $r \xleftarrow{\$} \mathbb{Z}_N$, compute and send $\mathcal{E}^2(\text{Enc}(c + r)) \leftarrow \mathcal{E}^2(\text{Enc}(c))^{\text{Enc}(r)}$ to S_2 .

3 **Server S_2 :**

4 Decrypt as $\text{Enc}(c + r)$ and send back to S_1

5 **Server S_1 :**

6 Receive $\text{Enc}(c + r)$ and compute: $\text{Enc}(c) = \text{Enc}(c + r) \cdot \text{Enc}(r)^{-1}$;

7 Output $\text{Enc}(c)$.

Note that nothing has been leaked to S_1 at the end of the protocol. However, there is some leakage function revealed to S_2 at current depth, which we will describe it in detail in later section. However, even by learning this pattern, S_2 has still no idea on which particular item is the same as the other at this depth since S_1 randomly permutes the item before sending to S_2 and everything has been encrypted. Moreover, no information has been leaked on the objects' scores.

8.2.2 Secure Best Score

The secure computation for the best score is different from computing the worst score. Below we describe the protocol SecBest between S_1 and S_2 :

Protocol 8.2. Server S_1 takes the inputs of the public key pk_p , $E(I) = \langle \text{EHL}(o), \text{Enc}(x) \rangle$ for the object o in list L_i , and a set of pointers $\mathcal{P} = \{j\}_{i \neq j, j \in \mathcal{M}}$ to the list in ER . Server S_2 's inputs are pk_p, sk_p . The protocol SecBest securely computes the encrypted best score at the current depth d , i.e., S_1 finally outputs $\text{Enc}(B(o))$, where $B(o)$ is the best score for the o at current depth.

At depth d , let $E(I)$ be the encrypted item in the list L_i , then its best score up to this depth is based on the whether this item has appeared in other lists $\{L_j\}_{j \neq i, j \in \mathcal{M}}$. The detailed description for SecBest is described in Algorithm 6.

In SecBest , S_1 has to scan the encrypted items in the other lists to securely evaluate the current best score for the encrypted $E(I)$. The last seen encrypted item in each sorted list contains the encryption

Algorithm 6: SecBest($E(I_i), \mathcal{P}, \text{pk}_p, \text{sk}_p$) Secure Best Score.

S_1 's input: $E(I_i)$ in list L_i , $\mathcal{P} = \{j\}_{i \neq j}$, pk_p
 S_2 's input: pk_p, sk_p

- 1 **Server S_1 :**
- 2 **foreach** list L_i **do**
- 3 maintain $\text{Enc}(\underline{x}_i^d)$ for L_i , where $\text{Enc}(\underline{x}_i^d)$ is the encrypted score at depth d .
- 4 Generate a random permutation $\pi : [l] \rightarrow [l]$;
- 5 Permute each L_i as $L_{\pi(i)} = \text{EHL}(o_{\pi(i)}), \text{Enc}(x_{\pi(i)})$;
- 6 **foreach** permuted $E(I_{\pi(i)})$ **do**
- 7 compute $\text{Enc}(b_i) \leftarrow \text{EHL}(o) \ominus \text{EHL}(o_i)$
- 8 send $\text{Enc}(b_i)$ to S_2 receive $\mathcal{E}^2(t_i)$ and compute:

$$\mathcal{E}^2(\text{Enc}(x'_i)) := \mathcal{E}^2(t_i)^{\text{Enc}(x_i)} \cdot \left(\mathcal{E}^2(1) \mathcal{E}^2(t_i)^{-1} \right)^{\text{Enc}(0)}$$
;
- 9 run $\text{Enc}(x'_i) \leftarrow \text{RecoverEnc}(\mathcal{E}^2(\text{Enc}(x'_i)), \text{pk}_p, \text{sk}_p)$ with S_2 ;
- 10 compute $\mathcal{E}^2(\text{Enc}(\underline{x}_i^d)) \leftarrow (1 - \prod_{i=1}^d \mathcal{E}^2(t_i))^{\text{Enc}(\underline{x}_i^d)}$;
- 11 run $\text{Enc}(\underline{x}_i^d) \leftarrow \text{RecoverEnc}(\mathcal{E}^2(\text{Enc}(\underline{x}_i^d)), \text{pk}_p, \text{sk}_p)$ with S_2 ;
- 12 set $\text{Enc}(B_i) \leftarrow \text{Enc}(\underline{x}_i^d) \cdot (\prod_{i=1}^l \text{Enc}(x'_i))$;
- 13 compute $\text{Enc}(B) \leftarrow \prod_{i=1}^m \text{Enc}(B_i)$ and output $\text{Enc}(B)$;
- 14 **Server S_2 :**
- 15 **for** $\text{Enc}(b_i)$ received from S_1 **do**
- 16 Decrypt to get b_i . If $b_i = 0$, set $t_i = 1$, otherwise, set $t_i = 0$;
- 17 Send $\mathcal{E}^2(t_i)$ to S_1 .

of the *best possible values* (or *bottom scores*). If the same object o appears in the previous depth then homomorphically adds the object's score to the encrypted best score $\text{Enc}(B)$, otherwise adds the bottom scores seen so far to $\text{Enc}(B)$. In particular, S_1 can homomorphically evaluate (at line 9): $\mathcal{E}^2(x'_i) = \mathcal{E}^2(t_i \cdot \text{Enc}(x_i) + (1 - t_i) \cdot \text{Enc}(0))$. That is, if $t_i = 0$ which means item I appeared in the previous depth, x'_i will be assigned the corresponding score x_i , otherwise, $x'_i = 0$. Similarly, S_1 homomorphically evaluates the following: $\text{Enc}(\underline{x}_i^d) = \text{Enc}((1 - \sum_i^d t_i) \cdot \underline{x}_i^d)$. If the item I does not appear in the previous depth, then $(1 - \sum_i^d t_i) = 1$ since each $t_i = 0$, therefore, \underline{x}_i^d will be assigned to the bottom value \underline{x}_i^d . Finally, S_1 homomorphically add up all the encrypted scores and get the encrypted best scores (line 12).

8.2.3 Secure Deduplication

At each depth, some of the objects might be repeatedly computed since the same objects may appear in different sorted list at the same depth. S_1 cannot identify duplicates since the items and their scores are probabilistically encrypted. We now present a protocol that deduplicates the encrypted objects in the following.

Protocol 8.3. Let the $\mathbf{E}(I)$ be an encrypted scored item such that $\mathbf{E}(I) = (\text{EHL}(o), \text{Enc}(W), \text{Enc}(B))$, i.e. the $\mathbf{E}(I)$ is associated with $\text{EHL}(o_i)$, its encrypted worst and best score $\text{Enc}(W_i), \text{Enc}(B_i)$. Assuming that S_1 's inputs are the public key pk_p , a set of encrypted scored items $Q = \{\mathbf{E}(I_i)\}_{i \in [|Q|]}$. Server S_2 has the public key pk_p and the secret key sk_p . The execution of the protocol SecDedup between S_1 and S_2 enables S_1 to get a new list of encrypted distinct objects and their scores, that is, at the end of the protocol, S_1 outputs a new list of items $\mathbf{E}(I'_1), \dots, \mathbf{E}(I'_l)$, and there does not exist $i, j \in [l]$ with $i \neq j$ such that $o_i = o_j$. Moreover, the new encrypted list should not affect the final top- k results.

Algorithm 7: SecDedup($Q = \{\mathbf{E}(I_i)\}_{i \in [|Q|]}, \text{pk}_p, \text{sk}_p$) : De-duplication Protocol

S_1 's input: $\mathbf{E}(I_1), \dots, \mathbf{E}(I_l)$, pk_p

S_2 's input: pk_p, sk_p

S_1 's output: Output $\mathbf{E}(I'_1) \dots \mathbf{E}(I'_l)$ without duplicated objects

```

1  Server  $S_1$ :
2    Let  $|Q| = l$ ;
3    for  $i = 1 \dots l$  do
4      for  $j = i + 1, \dots, l$  do
5        Compute  $\text{Enc}(b_{ij}) \leftarrow (\text{EHL}(o_i) \ominus \text{EHL}(o_j))$ ;
6      Set the symmetric matrix  $\mathbf{B}$  such that  $\mathbf{B}_{ij} = \text{Enc}(b_{ij})$ ;
7   $S_1$  generate it own public/private key  $(\text{pk}', \text{sk}')$ ;
8  for each  $\mathbf{E}(I_i)$  do
9    Generate random  $\alpha_i \in \mathbb{Z}_N^k$ ,  $\beta_i, \gamma_i \in \mathbb{Z}_N$ ;
10   Compute  $\mathbf{E}(\tilde{I}_i) = (\text{EHL}(\tilde{o}_i), \text{Enc}(\tilde{W}_i), \text{Enc}(\tilde{B}_i)) \leftarrow \text{Rand}(E(I_i), \alpha_i, \beta_i, \gamma_i)$ ;
11   Compute  $H_i = \text{Enc}_{\text{pk}'}(\alpha_i) || \text{Enc}_{\text{pk}'}(\beta_i) || \text{Enc}_{\text{pk}'}(\gamma_i)$  using  $\text{pk}'$ ;
12  Generate a random permutation  $\pi : [l] \rightarrow [l]$ ;
13  Permute  $\pi(\mathbf{B})$ , i.e. permute  $\mathbf{B}_{\pi(i)\pi(j)}$  for each  $B_{ij}$ ;
14  Permute  $\mathbf{E}(\tilde{I}_{\pi(i)})$  and  $H_{\pi(i)}$  for  $i \in [1, l]$ ;
15  Send  $\pi(\mathbf{B})$ ,  $\{\mathbf{E}(\tilde{I}_{\pi(i)})\}_{i=1}^l$ ,  $\{H_{\pi(i)}\}_{i=1}^l$ ,  $\text{pk}'$  to  $S_2$ ;

16 Server  $S_2$ :
17  Receive  $\pi(\mathbf{B})$ ,  $\{\mathbf{E}(\tilde{I}_{\pi(i)})\}_{i=1}^l$ ,  $\{H_{\pi(i)}\}_{i=1}^l$ , and  $\text{pk}'$  from  $S_1$ ;
18  for upper triangle of  $\pi(\mathbf{B})$  do
19    decrypt  $b_{\pi(i)\pi(j)} := \text{Dec}_{\text{sk}_p}(\mathbf{B}_{\pi(i)\pi(j)})$ ;
20    if  $b_{\pi(i)\pi(j)} = 0$  then
21      remove  $\mathbf{E}(\tilde{I}_{\pi(i)})$ ,  $H_{\pi(i)}$ ;
22      /* Deduplicate items */
23      randomly generate  $o_i$ , and  $\alpha_i \in \mathbb{Z}_N^k$ ,  $\beta_i, \gamma_i \in \mathbb{Z}_N$ ;
24      set  $W_i = Z + \beta_i$  and  $B_i = Z + \gamma_i$ , where  $Z = N - 1$ ;
25      Set  $\mathbf{E}(I'_{\pi(i)}) := (\text{EHL}(o_i) \odot \text{Enc}(\alpha_i), \text{Enc}(W_i), \text{Enc}(B_i))$ ;
26      Compute  $H'_{\pi(i)} \leftarrow \text{Enc}_{\text{pk}'}(\alpha_i) || \text{Enc}_{\text{pk}'}(\beta_i) || \text{Enc}_{\text{pk}'}(\gamma_i)$  using  $\text{pk}'$ ;

27  for remaining  $\text{Enc}(\tilde{I}_{\pi(j)})$ ,  $H_{\pi(j)}$  do
28    generate random  $\alpha'_i \in \mathbb{Z}_N^k$ ,  $\beta'_i$ , and  $\gamma'_i \in \mathbb{Z}_N$ ;
29     $\text{Enc}(I'_i) = (\text{EHL}(o'_i), \text{Enc}(W'_i), \text{Enc}(B'_i)) \leftarrow \text{Rand}(\text{Enc}(\tilde{I}_{\pi(j)}), \alpha'_i, \beta'_i, \gamma'_i)$ ;
30     $H_{\pi(j)} = \text{Enc}_{\text{pk}'}(\alpha_{\pi(j)}), \text{Enc}_{\text{pk}'}(\beta_{\pi(j)}), \text{Enc}_{\text{pk}'}(\gamma_{\pi(j)})$ ;
31    set  $H'_i = \text{Enc}_{\text{pk}'}(\alpha_{\pi(j)}) \cdot \text{Enc}_{\text{pk}'}(\alpha'_i) || \text{Enc}_{\text{pk}'}(\beta_{\pi(j)}) \cdot \text{Enc}_{\text{pk}'}(\beta'_i) || \text{Enc}_{\text{pk}'}(\gamma_{\pi(j)}) \cdot \text{Enc}_{\text{pk}'}(\gamma'_i)$ ;
32  Generate a random permutation  $\pi' : [l] \rightarrow [l]$ . Permute new list  $\mathbf{E}(I_{\pi'(i)})$  and  $H'_{\pi'(i)}$ , then send them
    back to  $S_1$ ;

33 Server  $S_1$ :
34  Decrypt each  $H'_{\pi'(i)}$  as  $\alpha'_{\pi'(i)}, \beta'_{\pi'(i)}, \gamma'_{\pi'(i)}$  using  $\text{sk}'$ ;
35  foreach  $\mathbf{E}(I'_{\pi'(i)}) = (\text{EHL}(o'_{\pi'(i)}), \text{Enc}(W'_{\pi'(i)}), \text{Enc}(B'_{\pi'(i)}))$  do
36    Run and get  $\text{Enc}(\hat{I}_i) = (\text{EHL}(\hat{o}_i), \text{Enc}(\hat{W}_i), \text{Enc}(\hat{B}_i)) \leftarrow \text{Rand}(\text{Enc}(I'_{\pi'(i)}), -\alpha'_{\pi'(i)}, -\beta'_{\pi'(i)}, -\gamma'_{\pi'(i)})$ ;
    Output the encrypted list  $\mathbf{E}(\hat{I}_1) \dots \mathbf{E}(\hat{I}_l)$ ;

```

Algorithm 8: Rand($\mathbf{E}(I), \alpha, \beta, \gamma$): Blinding the randomness

```
1 Let  $\mathbf{E}(I) = (\text{EHL}(o), \text{Enc}(B), \text{Enc}(W))$ ;  
2 Compute  $\mathbf{E}(\alpha), \text{Enc}(\beta), \text{Enc}(\gamma)$ ;  
3 Compute  $\text{EHL}(o) \leftarrow \text{EHL}(o) \odot \text{Enc}(\alpha)$ ,  $\text{Enc}(W) \leftarrow \text{Enc}(W) \cdot \text{Enc}(\beta)$ , and  $\text{Enc}(B) \leftarrow \text{Enc}(B) \cdot \text{Enc}(\gamma)$ ;  
4 Output  $\mathbf{E}(I') = (\text{EHL}(o), \text{Enc}(W), \text{Enc}(B))$ ;
```

Algorithm 9: A Secure Update Protocol SecUpdate($T^{d-1}, \Gamma^d, \text{pk}_p, \text{sk}_p$)

```
 $S_1$ 's input:  $\text{pk}_p, T^{d-1}, \Gamma^d$  (encrypted list without duplicated objects)  
 $S_2$ 's input:  $\text{pk}_p, \text{sk}_p$   
1 Server  $S_1$ :  
2   Permute  $\mathbf{E}(I_i) \in \Gamma^d$  as  $\mathbf{E}(I_{\pi(i)})$  based on random permutation  $\pi$ ;  
3   foreach each permute  $\mathbf{E}(I_{\pi(i)})$  do  
4     foreach each  $\mathbf{E}(I_j) \in T^{d-1}$  do  
5       Let  $\text{Enc}(W_i), \text{Enc}(B_i)$  be encrypted worst/best score in  $\mathbf{E}(I_{\pi(i)})$ , and let  $\text{Enc}(W_j), \text{Enc}(B_j)$  be  
6       encrypted worst/best score in  $\mathbf{E}(I_j)$ ;  
7       Compute  $\text{Enc}(b_{ij}) \leftarrow \text{EHL}(I_{\pi(i)}) \ominus \text{EHL}(I_j)$ , send  $\text{Enc}(b_{ij})$  to  $S_2$  and get  $\mathcal{E}^2(t_{ij})$ ;  
8       Compute  $\mathcal{E}^2(\text{Enc}(W'_i)) \leftarrow \mathcal{E}^2(t_{ij})^{\text{Enc}(W_i)}$ ,  $\text{Enc}(W'_i) \leftarrow \text{RecoverEnc}(\mathcal{E}^2(\text{Enc}(W'_i)), \text{pk}_p, \text{sk}_p)$ ,  
9        $\text{Enc}(W'_j) \leftarrow \text{Enc}(W_j)\text{Enc}(W'_i)$ ;  
10      Compute  $\mathcal{E}^2(\text{Enc}(B'_j)) \leftarrow \mathcal{E}^2(t_{ij})^{\text{Enc}(B_i)} (\mathcal{E}^2(1)\mathcal{E}^2(t_{ij})^{-1})^{\text{Enc}(B_j)}$   
11       $\text{Enc}(B'_j) \leftarrow \text{RecoverEnc}(\mathcal{E}^2(\text{Enc}(B'_j)), \text{pk}_p, \text{sk}_p)$ ;  
12      Set  $\text{Enc}(W'_j), \text{Enc}(B'_j)$  as the updated score for  $\text{Enc}(I_j)$ ;  
13      compute  $\mathcal{E}^2(\text{Enc}(W'_i)) \leftarrow \mathcal{E}^2(t_{ij})^{\text{Enc}(W_i)} (\mathcal{E}^2(1)\mathcal{E}^2(t_{ij})^{-1})^{\text{Enc}(W'_j)}$ , run  
14       $\text{Enc}(W'_i) \leftarrow \text{RecoverEnc}(\mathcal{E}^2(\text{Enc}(W'_i)), \text{pk}_p, \text{sk}_p)$  and maintain  $\text{Enc}(W'_i)$  for each  $\mathbf{E}(I_{\pi(i)})$   
15      Update the encrypted worst score to  $\text{Enc}(W'_i)$  for each  $\mathbf{E}(I_{\pi(i)})$  and keep the original best score  
16       $\text{Enc}(B_i)$ ;  
17      Append the updated  $\text{Enc}(I_{\pi(i)})$  to  $T^{d-1}$  and get  $T^d$ ;  
18       $S_1$  and  $S_2$  execute SecDedup( $T^d, \text{pk}_p, \text{sk}_p$ ) and get the updated list  $T^d$ ;  
19       $S_1$  finally outputs  $T^d$ .  
20 Server  $S_2$ :  
21   foreach  $\text{Enc}(b_i)$  received from  $S_1$  do  
22     Decrypt to get  $b_i$ ;  
23     If  $b_i = 0$ , set  $t_i = 1$ , otherwise, set  $t_i = 0$ . Send  $\mathcal{E}^2(t_i)$  to  $S_1$ .
```

Intuitively, at a high level, SecDedup let S_2 obviously find the duplicated objects and its scores, and replaces the object id with a random value and its score with a large enough value $Z = N - 1 \in \mathbb{Z}_N$ (the largest value in the message space) such that, after sorting the worst scores, it will definitely not appear in the top- k list.

Figure 3 gives the overview of our approach. The technical challenge here is to allow S_2 to find the duplicated objects without letting S_1 know which objects have been changed. The idea is to let the server S_1 send a encrypted permuted matrix \mathbf{B} , which describes the pairwise equality relations between the objects in the list. S_1 then use the same permutation to permute the list of blinded encrypted items before sending it to S_2 . This prevents S_2 from knowing the original data. For the duplicated objects, S_2 replace the scores with a large enough encrypted worst score. On the other hand, after deduplication, S_2 also has to blind the data items as well to prevent S_1 from knowing which items are the duplicated ones. S_1 finally gets the encrypted items without duplication. Algorithm 7 describes the detailed protocol.

We briefly discuss the execution of the protocol as follows: S_1 first fill the entry \mathbf{B}_{ij} by computing

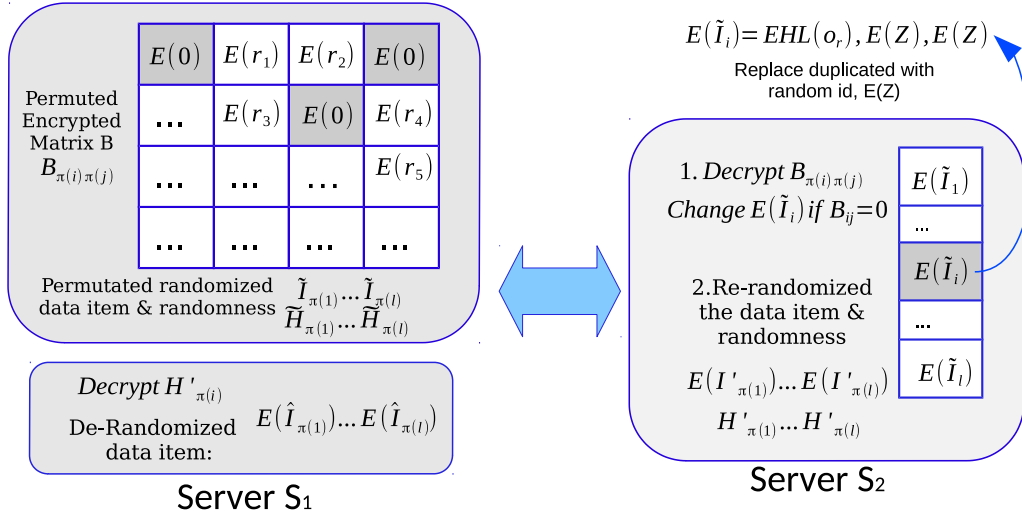


Figure 3: Overview of the SecDedup protocol

$EHL(o_i) \ominus EHL(o_j)$. Note that, since the encrypted \mathbf{B} is symmetric matrix indicating the equality relations for the list, therefore, S_1 only need fill the upper triangular for \mathbf{B} . and lower triangular can be filled by the fact that $\mathbf{B}_{ij} = \mathbf{B}_{ji}$. In addition, S_1 blinds the encrypted item $\text{Enc}(I_i)$ by homomorphically adding random values and get $\text{Enc}(\tilde{I}_i)$. This prevents S_2 from knowing the values of the item since S_2 has the secret key. Moreover, S_1 encrypts the randomnesses using his own public key pk' and get H_i . To hide the relation pattern between the objects in the list, S_1 applies a random permutation π to the matrix $\mathbf{B}_{\pi(i)\pi(j)}$, as well as $\text{Enc}(I_{\pi(i)})$ and $H_{\pi(i)}$. Receiving the ciphertext, S_2 only needs to decrypt the upper triangular of the matrix, S_2 only keeps one copy of the $\text{Enc}(\tilde{I}_{\pi(i)})$, $H_{\pi(i)}$ and $\text{Enc}(\tilde{I}_{\pi(j)})$, $H_{\pi(j)}$ if $b_{\pi(i)\pi(j)} = 0$. Without loss of generality, we keep $\text{Enc}(\tilde{I}_{\pi(j)})$, $H_{\pi(j)}$ and replace $\text{Enc}(\tilde{I}_{\pi(i)})$, $H_{\pi(i)}$ as line 22-25. For the unchanged item, S_2 blinds them using as well (see line 28-30). It worth noting that the randomnesses added by S_2 are to prevent S_1 from discovering which item has been changed or not. S_2 also randomly permute the list as well (line 31). S_1 homomorphically recovers the original values by decrypting the received $H'_{\pi'(i)}$ using his sk' (see line 35). S_1 eventually the new permuted list of encrypted items.

For the duplicated objects, the protocol replaces their object id with a random value, and its worst score with a large number Z . For the new encrypted items that S_2 replaced (line 22), $\text{Enc}(\tilde{I}_i) = (EHL(\hat{o}_i), \text{Enc}(\tilde{W}_i), \text{Enc}(\tilde{B}_i))$, we show in the following that $\text{Enc}(\tilde{W}_i)$ is indeed a new encryption of the permuted $\text{Enc}(W_{\pi'(\pi(j))})$ for some $j \in [l]$. As we can see, the $\text{Enc}(\tilde{W}_i)$ is permuted by S_2 's random π' , i.e. $\text{Enc}(\tilde{W}_{\pi'(i)})$ (see line 31). Hence, it follows that:

$$\text{Enc}(\tilde{W}_{\pi'(i)}) \sim \text{Enc}(W'_{\pi'(i)} - \beta'_{\pi'(i)}) \quad (2)$$

$$\sim \text{Enc}(W'_{\pi'(i)} - (\beta_{\pi'(\pi(j))} + \beta'_{\pi'(i)})) \quad (3)$$

$$\sim \text{Enc}(\tilde{W}_{\pi'(\pi(j))} + \beta_{\pi'(\pi(j))} - (\beta_{\pi'(\pi(j))} + \beta'_{\pi'(i)})) \quad (4)$$

$$\sim \text{Enc}(W_{\pi'(\pi(j))} + \beta_{\pi'(\pi(j))} + \beta'_{\pi'(i)} - (\beta_{\pi'(\pi(j))} + \beta'_{\pi'(i)})) \quad (5)$$

$$\sim \text{Enc}(W_{\pi'(\pi(j))}) \quad (6)$$

In particular, from Algorithm 7, we can see that Equation (2) holds due to line 35, Equation (3) holds since line 30 and 33, Equation (4) holds due to line 28, and Equation (5) holds because of line 10. On the other hand, for the duplicated items that S_1 has changed from line 22 to 25, by the homomorphic operations of S_1 at line 35, we have

$$\text{Enc}(\tilde{W}_{\pi'(k)}) \sim \text{Enc}(W'_{\pi'(k)} - \beta'_{\pi'(k)}) \sim \text{Enc}(Z + \beta'_{\pi'(k)} - \beta'_{\pi'(k)}) \sim \text{Enc}(Z)$$

Since Z is a very large enough number, this randomly generated objects definitely do not appear in the top- k list after sorting.

8.2.4 Secure Update

At each depth d , we need to update the current list of objects with the latest global worst/best scores. At a high level, S_1 has to update the encrypted list Γ^d from the state T^{d-1} (previous depth) to T^d , and appends the new encrypted items at this depth. Let Γ^d be the list of encrypted items with the encrypted worst/best scores S_1 get at depth d . Specifically, for each encrypted item $E(I_i) \in T^{d-1}$ and each $E(I_j) \in L_d$ at depth d , we update I_i 's worst score by adding the worst from I_j and replace its best score with I_j 's best score if $I_i = I_j$ since the worst score for I_j is the in-depth worst score and best score for I_j is the most updated best score. If $I_i \neq I_j$, we then simply append $E(I_j)$ with its scores to the list. Finally, we get the fresh T^d after depth d . We describe the SecUpdate protocol in Algorithm 9.

9 Security

Since our construction supports a more complex query type than searching, the security has to capture the fact that the adversarial servers also get the ‘views’ from the data and meta-data during the query execution. The CQA security model in our top- k query processing defines a **Real** world and an **Ideal** world. In the real world, the protocol between the adversarial servers and the client executes just like the real SecTopK scheme. In the ideal world, we assume that there exists two simulator Sim_1 and Sim_2 who get the leakage profiles from an ideal functionality and try to simulate the execution for the real world. We say the scheme is CQA secure if, after polynomial many queries, no ppt distinguisher can distinguish between the two worlds only with non-negligibly probability. We give the formal security definition in Definition 9.1.

Definition 9.1. Let $\text{SecTopK} = (\text{Enc}, \text{Token}, \text{SecQuery})$ be a top- k query processing scheme and consider the following probabilistic experiments where \mathcal{E} is an environment, \mathcal{C} is a client, S_1 and S_2 are two non-colluding semi-honest servers, Sim_1 and Sim_2 are two simulators, and $\mathcal{L}_{\text{Setup}}, \mathcal{L}_{\text{Query}} = (\mathcal{L}_{\text{Query}}^1, \mathcal{L}_{\text{Query}}^2)$ are (stateful) leakage functions:

Ideal(1^λ): The environment \mathcal{E} outputs a relation R of size n and sends it to the client \mathcal{C} . \mathcal{C} submits the relation R to $\mathcal{F}_{\text{topk}}$, i.e. an **ideal** top- k functionality. $\mathcal{F}_{\text{topk}}$ outputs $\mathcal{L}_{\text{Setup}}(R)$ and 1^λ , and gives $\mathcal{L}_{\text{Setup}}(R)$, 1^λ to Sim_1 . Given $\mathcal{L}_{\text{Setup}}(R)$ and 1^λ , Sim_1 generates an encrypted ER.

\mathcal{C} generates a polynomial number of adaptively chosen queries (q_1, \dots, q_m) . For each q_i , \mathcal{C} submits q_i to $\mathcal{F}_{\text{topk}}$, $\mathcal{F}_{\text{topk}}$ then sends $\mathcal{L}_{\text{Query}}^1(\text{ER}, q_i)$ to Sim_1 and sends $\mathcal{L}_{\text{Query}}^2(\text{ER}, q_i)$ to Sim_2 .

After the execution of the protocol, \mathcal{C} outputs $\text{OUT}'_{\mathcal{C}}$, Sim_1 outputs $\text{OUT}_{\text{Sim}_1}$, and Sim_2 outputs $\text{OUT}_{\text{Sim}_2}$.

Real_A(1^λ): The environment \mathcal{E} outputs a relation R of size n and sends it to \mathcal{C} . \mathcal{C} computes $(K, \text{ER}) \leftarrow \text{Enc}(1^\lambda, R)$ and sends the encrypted ER to S_1 .

\mathcal{C} generates a polynomial number of adaptively chosen queries (q_1, \dots, q_m) . For each q_i , \mathcal{C} computes $\text{tk}_i \leftarrow \text{Token}(K, q_i)$ and sends tk_i to S_1 . S_1 run the protocol $\text{SecQuery}(\text{tk}_i, \text{ER})$ with S_2 .

After the execution of the protocol, S_1 sends the encrypted results to \mathcal{C} . \mathcal{C} outputs $\text{OUT}_{\mathcal{C}}$, S_1 outputs OUT_{S_1} , and S_2 outputs OUT_{S_2} .

We say that SecQuery is adaptively $(\mathcal{L}_{\text{Setup}}, \mathcal{L}_{\text{Query}})$ -semantically secure (CQA) if the following holds:

1. For all \mathcal{E} , for all S_1 , there exists a ppt simulator Sim_1 such that the following two distribution ensembles are computationally indistinguishable

$$\langle \text{OUT}_{S_1}, \text{OUT}_{\mathcal{C}} \rangle \cong \langle \text{OUT}_{\text{Sim}_1}, \text{OUT}'_{\mathcal{C}} \rangle$$

2. For all \mathcal{E} , for all S_2 , there exists a ppt simulator Sim_2 such that the following two distribution ensembles are computationally indistinguishable

$$\langle \text{OUT}_{S_2}, \text{OUT}_{\mathcal{C}} \rangle \cong \langle \text{OUT}_{\text{Sim}_2}, \text{OUT}'_{\mathcal{C}} \rangle$$

We formally define the leakage function in **SecTopK**. Let the setup leakage $\mathcal{L}_{\text{Setup}} = (|R|, |M|)$, i.e. the size of the database and the total number of attributes. $\mathcal{L}_{\text{Setup}}$ is the leakage profile revealed to S_1 after the execution of **Enc**. During the query processing, we allow $\mathcal{L}_{\text{Query}} = (\mathcal{L}_{\text{Query}}^1, \mathcal{L}_{\text{Query}}^2)$ revealed to the servers. Note that $\mathcal{L}_{\text{Query}}^1$ is the leakage function for S_1 , while $\mathcal{L}_{\text{Query}}^2$ is the leakage function for S_2 . In our scheme, $\mathcal{L}_{\text{Query}}^1 = (\text{QP}, D_q)$, where **QP** is the *query pattern* indicating whether a query has been repeated or not. Formally, for $q_j \in \mathbf{q}$, the *query pattern* $\text{QP}(q_j)$ is a binary vector of length j with a 1 at location i if $q_j = q_i$ and 0 otherwise. D_q is the halting depth for query q . For any query q , we define the equality pattern as follows: suppose that there are m number of objects at each depth, then

- *Equality pattern* $\text{EP}_d(q)$: a symmetric binary $m \times m$ matrix M^d , where $M^d[i, j] = 1$ if there exist $o_{\pi(i')} = o_{\pi(j')}$ for some random permutation π such that $\pi(i') = i$ and $\pi(j') = j$, otherwise $M^d[i, j] = 0$.

Then, let $\mathcal{L}_{\text{Query}}^2 = (\{\text{EP}_d(q)\}_{d=1}^{D_q})$, i.e. at depth $d \leq D_q$ the equality pattern indicates the number of equalities between objects. Note that $\text{EP}^d(q)$ does not leak the equality relations between objects at any depth in the original database, i.e. the server never knows which objects are same since the server doesn't know the permutation.

Theorem 9.2. *Suppose the function used in EHL is a pseudo-random function and the Paillier encryption is CPA-secure, then the scheme **SecTopK** = (Enc, Token, SecQuery) we proposed is $(\mathcal{L}_{\text{Setup}}, \mathcal{L}_{\text{Query}})$ -CQA secure.*

Proof: We describe the ideal functionality $\mathcal{F}_{\text{topk}}$ as follows. An environment \mathcal{E} samples samples $\phi_i \xleftarrow{\$} \{0, 1\}^{\log n}$ for $1 \leq i \leq n$ to create a set Φ of n distinct identifiers for n objects. For each object, \mathcal{E} randomly pick M attribute values from \mathbb{Z}_N . \mathcal{E} finally outputs the relation R with the sampled n objects associated with M attributes. \mathcal{E} sends the R to the client \mathcal{C} , and \mathcal{C} outsources the relation to the ideal functionality $\mathcal{F}_{\text{topk}}$. During the setup phase, $\mathcal{F}_{\text{topk}}$ computes $\mathcal{L}_{\text{Setup}} = (n, M)$, where n is the number of the objects and M is the number of the attributes. Then $\mathcal{F}_{\text{topk}}$ sends $\mathcal{L}_{\text{Setup}}$ to a simulator Sim_1 , which we'll describe it next. Sim_1 receives $\mathcal{L}_{\text{Setup}}$, then Sim_1 generates the random keys $\kappa_1, \dots, \kappa_s$ for the EHL and pk_p, sk_p for the paillier encryption. Then Sim_1 encrypts each object using EHL and each score using the paillier encryption.

The client adaptively generate a number of queries $q_1 \dots q_m$. For each q_i , $\mathcal{F}_{\text{topk}}$ computes $\mathcal{L}_{\text{Query}}^1 = (\text{QP}, D_{q_i})$ and $\mathcal{L}_{\text{Query}}^2 = (\{\text{EP}_d(q_i)\}_{d=1}^{D_{q_i}})$. $\mathcal{F}_{\text{topk}}$ sends $\mathcal{L}_{\text{Query}}^1$ to Sim_1 and sends $\mathcal{L}_{\text{Query}}^2 = (\{\text{EP}_d(q_i)\}_{d=1}^{D_{q_i}})$ to Sim_2 . Next, we describe Sim_1 and Sim_2 . As mentioned above, Sim_1 learns the leakage function $\mathcal{L}_{\text{Query}}^1 = (\text{QP}, D_{q_i})$. Given the leakage **QP**, Sim_1 first checks if either of the query q_i appeared in any previous query. If q_i appeared previously, Sim_1 runs the same simulation of **SecQuery** as before. If not, Sim_1 invokes the simulations for each of the sub-routine from **SecQuery**. We show in the Appendix A (See Definition A.1 and Lemma A.2) that our building blocks are secure. Therefore, Sim_1 can invoke the simulation the original **SecQuery** by calling the underlying simulators Sim_1 from those sub-protocols **SecWorst**, **SecBest**, **SecDedup**, and **SecUpdate**. As those sub-protocols have been proved to be secure, the Sim_1 can simulate the execution of the original **SecQuery**. Moreover, the messages sent during the execution are either randomly permuted or protected by the semantic encryption scheme. Therefore, at the end of the protocol, $\text{OUT}_{\text{Sim}_1}$ looks indistinguishable from the the output OUT_{S_1} , i.e.

$$\langle \text{OUT}_{S_1}, \text{OUT}_{\mathcal{C}} \rangle \cong \langle \text{OUT}_{\text{Sim}_1}, \text{OUT}'_{\mathcal{C}} \rangle$$

By knowing $\mathcal{L}_{\text{Query}}^2$, Sim_2 can simulate the execution of the original SecQuery . Similarly, Sim_2 needs to call the underlying the simulator Sim_2 from the building blocks SecWorst , SecBest , SecDedup , and SecUpdate . As the messages sent during the execution are either randomly permuted or protected by the semantic encryption scheme, Sim_2 learns nothing except the leakage $\mathcal{L}_{\text{Query}}^2$. Therefore, at the end of the protocol, $\text{OUT}_{\text{Sim}_2}$ looks indistinguishable from the the output OUT_{S_2} ,

$$\langle \text{OUT}_{S_2}, \text{OUT}_C \rangle \cong \langle \text{OUT}_{\text{Sim}_2}, \text{OUT}'_C \rangle$$

■

10 Query Optimization

In this section, we present some optimizations that improve the performance of our protocol. The optimizations are two-fold: 1) we optimize the efficiency of the protocol SecDedup at the expense of some additional privacy leakage, and 2) we propose batch processing of SecDupElim and EncSort to further improve the SecQuery .

10.1 Efficient SecDupElim

We now introduce the efficient protocol SecDupElim that provides similar functionality as SecDedup . Recall that, at each depth, S_1 runs SecDedup to deduplicate m encrypted objects, then after the execution of SecDedup S_1 still receives m items but without duplication, and add these m objects to the list T^d when running SecUpdate . Therefore, when we execute the costly sorting algorithm EncSort the size of list to sort has md elements at depth d .

The idea for SecDupElim is that instead of keeping the same number encrypted items m , SecDupElim *eliminates* the duplicated objects. In this way, the number of encrypted objects gets reduced, especially if there are many duplicated objects. The SecDupElim can be obtained by simply changing the SecDedup as follows: in Algorithm 7 at line 20, when S_2 observes that there exist duplicated objects, S_2 only keeps one copy of them. The algorithm works exactly the same as before but without performing the line 22-25. We also run SecDupElim instead of SecDedup at line 13 in the SecUpdate . That is, after secure update, we only keep the distinct objects with updated scores. Thus, the number of items to be sorted also decrease. Now by adapting SecDupElim , if there are many duplicated objects appear in the list, we have much fewer encrypted items to sort.

Remark on security. The SecDupElim leaks additional information to the server S_1 . S_1 learns the *uniqueness pattern* $\text{UP}^d(q_i)$ at depth d , where $\text{UP}^d(q_i)$ denotes the number of the unique objects that appear at current depth d . The distinct encrypted values at depth d are independent from all other depths, therefore, this protocol still protects the distribution of the original ER. In addition, due to the ‘re-encryptions’ during the execution of the protocol, all the encryptions are fresh ones, i.e., there are not as the same as the encryptions from ER. Finally, we emphasize that nothing on the objects and their values have been revealed since they are all encrypted.

10.2 Batch Processing for SecQuery

In the query processing SecQuery , we observe that we do not need to run the protocols SecDupElim and EncSort for every depth. Since SecDupElim and EncSort are the most costly protocols in SecQuery , we can perform *batch processing* and execute them after a few depths and not at each depth. Our observation is that there is no need to deduplicate repeated objects at each scanned depth. If we perform the SecDupElim after certain depths of scanning, then the repeated objects will be eliminated, and those distinct encrypted objects with updated worst and best scores will be sorted by running EncSort . The protocol will remain correct. We introduce a parameter p such that $p \geq k$. The parameter p specifies

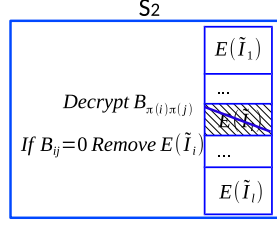


Figure 4: SecDupElim

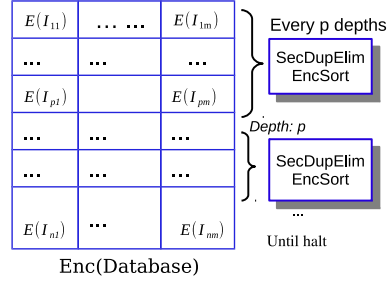


Figure 5: Batching Process

where we need to run the SecDupElim and EncSort in the SecQuery protocol. That is, the server S_1 runs the SecQuery with S_2 the same as in Algorithm 3, except that every p depths we run line 9-12 in Algorithm 3 to check if the algorithm could halt. In addition, we can replace the SecDupElim with the original SecDedup in the batch processing for better privacy but at the cost of some efficiency.

Security. Compared to the optimization from SecDupElim, we show that the batching strategy provides more privacy than just running the SecDupElim alone. For query q , assuming that we compute the scores over m attributes. Recall that the $UP^p(q)$ at depth p has been revealed to S_1 while running SecDupElim, therefore, after the first depth, in the worst case, S_1 learns that the objects at the first depth is the same object. To prevent this worst case leakage, we perform SecDupElim every p depth. Then S_1 learns there are p distinct objects in the worst case. After depth p , the probability that S_1 can correctly locate those distinct encrypted objects' positions in the table is at most $\frac{1}{(p!)^m}$. This decreases fast for bigger p . However, in practice this leakage is very small as many distinct objects appear every p depth. Similar to all our protocols, the encryptions are fresh due to the 're-encryption' by the server. Even though S_1 has some probability of guessing the distinct objects' location, the object id and their scores have not been revealed since they are all encrypted.

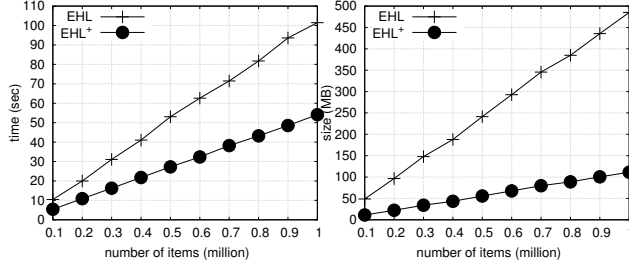
10.3 Efficiency

We analyze the efficiency of query execution. Suppose the client chooses m attributes for the query, therefore at each depth there are m objects. At depth d , it takes S_1 $O(m)$ for executing SecWorst, $O(md)$ for executing SecBest, $O(m^2)$ for SecDedup, and $O(m^2d)$ for the SecUpdate. The complexities for S_2 are similar. In addition, the EncSort has time overhead $O(m \log^2 m)$; however, we can further reduce to $O(\log^2 m)$ by adapting parallelism (see [7]). On the other hand, the SecDupElim only takes $O(u^2)$, where u is the number of distinct objects at this depth. Notice that most of the computations are multiplication (homomorphic addition), therefore, the cost of query processing is relatively small.

11 Experiments

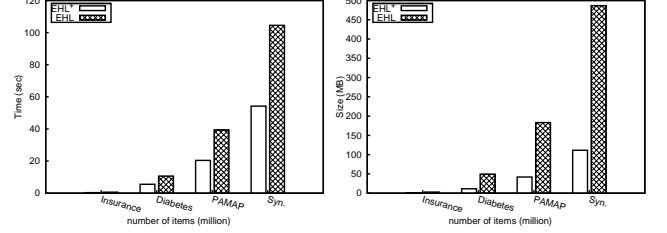
To evaluate the performance of our protocols, we conducted a set of experiments using real and synthetic datasets. We used the HMAC-SHA-256 as the pseudo-random function (PRF) for the EHL and EHL⁺ encoding, a 128-bit security for the Pailliar and DJ encryption, and all experiments are implemented using C++. We implement the scheme SecTopK = (Enc, Token, SecQuery), including all the protocols SecWorst, SecBest, EncSort, and EncCompare and their optimizations. We run our experiments on a 24 core machine, who serves as the cloud, running Scientific Linux with 128GB memory and 2.9GHz Intel Xeon.

DataSets We use the following real world dataset downloaded from UCI Machine Learning Repository [34]. **insurance:** a benchmark dataset that contains 5822 customers' information on an insurance



(a) Construction Time

(b) Size Overhead

Figure 6: Encryption using EHL vs. EHL⁺.

(a) Construction Time

(b) Size Overhead

Figure 7: Encryption EHL vs. EHL⁺ on real data

company and we extracted 13 attributes from the original dataset. **diabetes**: a patients' dataset containing 101767 patients' records (i.e. data objects), where we extracted 10 attributes. **PAMAP**: a physical activity monitoring dataset that contains 376416 objects, and we extracted 15 attributes. We also generated synthetic datasets **synthetic** with 10 attributes that takes values from Gaussian distribution and the number of records are varied between 5 thousands to 1 million.

11.1 Evaluation of the Encryption Setup

We implemented both the EHL and the efficient EHL⁺. For EHL, to minimize the false positives, we set the parameters as $H = 23$ and $s = 5$, where L is the size of the EHL and s is the number of the secure hash functions. For EHL⁺, we choose the number of secure hash function HMAC in EHL⁺ to be $s = 5$, and, as discussed in the previous section, we obtained negligible false positive rate in practice. The encryption Enc is independent of the characteristics of the dataset and depends only on the size. Thus, we generated datasets such that the number of the objects range from 0.1 to 1 million. We compare the encryptions using EHL and EHL⁺. After sorting the scores for each attribute, the encryption for each item can be fully parallelized. Therefore, when encrypting each dataset, we used 64 threads on the machine that we discussed before. Figure 6 shows that, both in terms of time and space, the cost of database encryption Enc is reasonable and scales linearly to the size of the database. Clearly, EHL⁺ has less time and space overhead. For example, it only takes 54 seconds to encrypt 1 million records using EHL⁺. The size is also reasonable, as the encrypted database only takes 111 MB using EHL⁺. Figure 7 also shows the encryption time and size overhead for the real dataset that we used. Finally, we emphasize that the encryption only incurs a one-time off-line construction overhead.

11.2 Query Processing Performance

11.2.1 Query Performance and Methodology

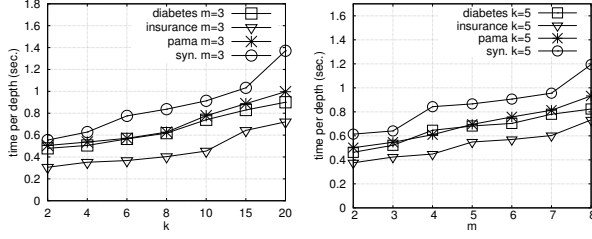
We evaluate the performance of the secure query processing and their optimizations that we discussed before. In particular, we use the query algorithm without any optimization but with full privacy, denoted as Qry.F; the query algorithm running SecDupElim instead of SecDedup at every depth, denoted as Qry.E; and the one using the batching strategies, denoted as Qry.Ba. We evaluate the query processing performance using all the datasets and use EHL⁺ to encrypt all of the object ids.

Notice that the performance of the NRA algorithm depends on the distribution of the dataset among other things. Therefore, to present a clear and simple comparison of the different methods, we measure the average time per depth for the query processing, i.e. $\frac{T}{D}$, where T is the total time that the program spends on executing a query and D is the total number of depths the program scanned before halting. In most of our experiments the value of D ranges between a few hundred and a few thousands. For each query, we randomly choose the number of attributes m that are used for the ranking function

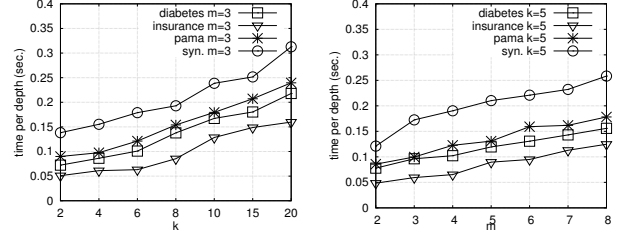
ranging from 2 to 8, and we also vary k between 2 and 20. The ranking function F that we use is the sum function.

11.2.2 Qry_F evaluation

We report the query processing performance without any query optimization. Figure 8 shows Qry_F query performance. The results are very promising considering that the query is executed completely on encrypted data. For a fixed number of attributes $m = 3$, the average time is about 1.30 seconds for the largest dataset **synthetic** running top-20 queries. When fixing $k = 5$, the average time per depth for all the dataset is below 1.20 seconds. As we can see that, for fixed m , the performance scales linearly as k increases. Similarly, the query time also linearly increases as m gets larger for fixed k .



(a) Performance varying k (b) Performance varying m



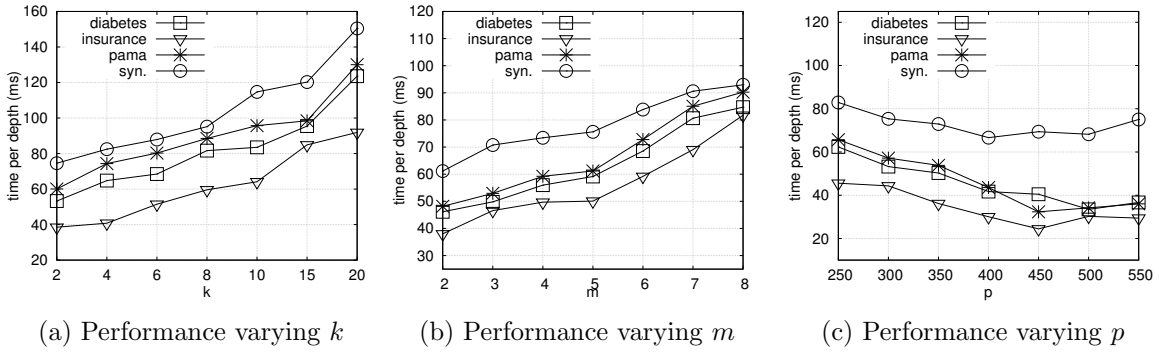
(a) Performance varying k (b) Performance varying m

Figure 8: Qry_F query performance

Figure 9: Qry_E query optimization performance

11.2.3 Qry_E evaluation

The experiments show that the SecDupElim improves the efficiency of the query processing. Figure 9 shows the querying overhead for exactly the same setting as before. Since Qry_E eliminates all the duplicated the items for each depth, Qry_E has been improved compared to the Qry_F above. As k increases, the performance for Qry_E executes up to 5 times faster than Qry_F when k increase to 20. On the other hand, fixing $k = 5$, the performance of Qry_E can execute up to around 7 times faster than Qry_F as m grows to 20. In general, the experiments show that Qry_E effectively speed up the query time 5 to 7 times over the basic approach.



(a) Performance varying k

(b) Performance varying m

(c) Performance varying p

Figure 10: Qry_Ba query optimization performance

11.2.4 Qry_Ba evaluation

We evaluate the effectiveness of batching optimization for the Qry_Ba queries. Figure 10 shows the query performance of the Qry_Ba for the same settings as the previous experiments. The experiments

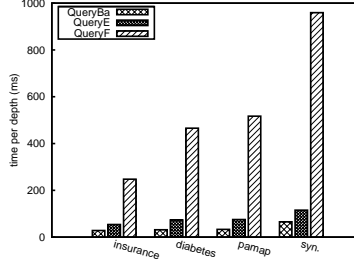


Figure 11: Comparisons ($k = 5$, $m = 2$, and $p = 500$)

show that the batching technique further improves the performance. In particular, for fixed batching parameter $p = 150$, i.e. every 150 depths we perform **SecDupElim** and **EncSort** in the **SecQuery**, and we vary our k from 2 to 20. Compared to the **Qry_E**, the average time per depth for all of the datasets have been further improved. For example, when $k = 2$, the average time for the largest dataset **synthetic** is reduced to 74.5 milliseconds, while for **Qry_F** it takes more than 500 milliseconds. For **diabetes**, the average time is reduced to 53 milliseconds when $k = 2$ and 123.5 milliseconds when k increases to 20. As shown in figure 10a, the average time linearly increases as k gets larger. Similarly, when fixing the $k = 5$ and $p = 150$, for **synthetic** the performance per depth reduce to 61.1 milliseconds and 92.5 milliseconds when $m = 2$ and 8 separately. In Figure 10c, We further evaluate the parameter p . Ranging p from 200 to 550, the experiments show that the proper p can be chosen for better query performance. For example, the performance for **diabetes** achieves the best when $p = 450$. In general, for different dataset, there are different p 's that can achieve the best query performance. When p gets larger, the number of calls for **EncSort** and **SecDupElim** are reduced, however, the performance for these two protocols also slow down as there're more encrypted items.

We finally compare the three queries' performance. Figure 11 shows the query performance when fixing $k = 5$, $m = 3$, and $p = 500$. Clearly, as we can see, **Qry_Ba** significantly improves the performance compared to **Qry_F**. For example, compared to **Qry_F**, the average running time is roughly 15 times faster for **PAMAP**.

11.2.5 Communication Bandwidth

We evaluate the communication cost of our protocol. *Our experiments show that the network latency is significantly less than the query computation cost.* In particular, we evaluate the communication of the fully secure and un-optimized **Qry_F** queries on the largest dataset **synthetic**. For each depth, the bandwidth is the same since the duplicated encrypted objects are filled with encryptions of random values. Each ciphertext is 32 bytes and each round only a few ciphertexts are transferred. Especially, if we use m attributes in our query we communicate between m and m^2 number of ciphertexts each time. So, for $m = 4$, we use between 1024 to 4096 bytes messages. Also, the number of messages per depth (per step) is 12. So, in the worst case, we need to submit 12 total messages between S_1 and S_2 of 4KB each.

We evaluate the bandwidth on the largest dataset **synthetic**. Note that, the bandwidth per depth is independent of k since each depth this communication size only depends on m . As mentioned the bandwidth is $O(m^2)$, by varying m we show in Figure 12a the bandwidth per depth. In Figure 12b, we show the total bandwidth when executing the top-20 by fixing $m = 4$. As we can see, the total size of the bandwidth is very small, therefore, the total latency could be very small for a high-speed connection between the two clouds. The speed of the network between two clouds depends on the location and the technology of the clouds. A recent study showed that we can achieve more than 70 Mbps for two clouds where one is in the US and the other in Japan [19]. Furthermore, with recent networking advances⁷,

⁷<http://www.cisco.com/c/en/us/products/cloud-systems-management/intercloud-fabric/index.html>

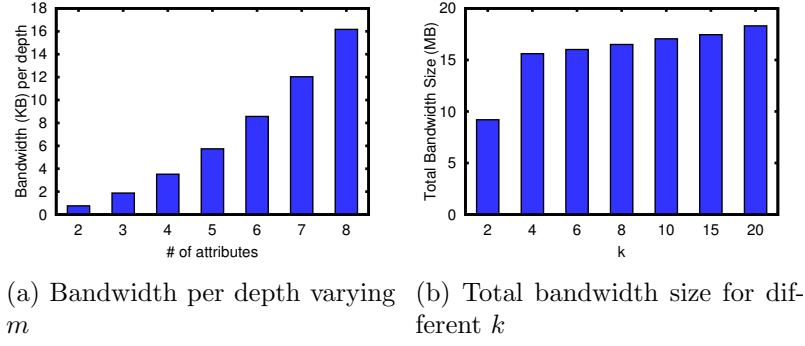


Figure 12: Communication bandwidth evaluation

Dataset	bandwidth (MB)	latency (sec.)
insurance	8.87	1.41
diabetes	12.45	1.99
PAMAP	15.72	2.5152
synthetic	17.3	2.768

Table 3: Comm. bandwidth & latency ($k = 20$, $m = 4$)

we expect that the connections between clouds (inter-clouds) will be much higher [8]. However, even if we assume that the communication between the two clouds is about 50 Mbps, the total cost of the communication at each depth is below 1 ms! Thus, communication is not a bottleneck for our protocol. In Figure 12a, we report the actual bandwidth per depth. In Figure 12b, we show the total bandwidth when executing the top-20 by fixing $m = 4$. As we can see, the total size of the bandwidth is very small that confirms our intuition. Also, assuming a standard 50 Mbps LAN setting, we show in Table 3 the total network latency between servers S_1 and S_2 when $k = 20$ and $m = 4$. Based on the discussion above, we can see that the communication cost of our protocol is very moderate for any reasonable assumptions about the connectivity between the two clouds. The total cost of our protocol is dominated by the computational cost that was presented in the previous section.

12 Top-k Join

We would like to briefly mention that our technique can be also extended to compute top- k join queries over multiple encrypted relations. Given a set of relations, R_1, \dots, R_L , each tuple in R_i is associated with some score that gives it a rank within R_i . The *top-k join query* joins R_1 to R_L and produces the results ranked on a total score. The total score is computed according to some function, F , that combines individual scores. We consider only (i.e. *equi-join*) conditions in this paper. Similarly, the score function F we consider in this paper is also a linear combination over the attributes from the joining relations. A possible SQL-like join query example is as follows: `Q1 = SELECT * FROM A,B,C WHERE A.1=B.1 and B.2=C.3 ORDER BY A.1+B.2+C.4 STOP AFTER k;` where A, B and C are three relations and A.1, B.1, B.2, C.3, C.4 are attributes on these relations. Our idea is to design a secure join operator, denoted as \bowtie_{sec} , such that the server S_1 obviously joins the relations based on the received token. S_1 has to invoke a protocol with S_2 to get the resulting joined results that meet the join condition.

12.1 Secure Top-k Join

We provide a description of the secure top- k join in this section. Since a join operator is implemented in most system as a dyadic (2-way) operator, we describe the secure top- k operator as a *binary* join operator between two relations R_1 and R_2 . Consider an authorized client that wants to join two

encrypted relations and get the top- k based on a join condition. Assume that each tuple in R_i has m_i many attributes and each R_i have n_i many tuples for $i = \{1, 2\}$. Furthermore, denote o_j^i be the j th objects in R_i and let $o_j^i.x_k$ be the k th attribute value.

12.2 Encryption Setup for Multiple databases

Algorithm 10: $\text{Enc}(R_1, R_2)$: database encryption

- 1 Generate public/secret key pk_p, sk_p for the pailliar encryption, generate random secret keys $\kappa_1, \dots, \kappa_s$ for the EHL;
 - 2 **foreach** each $o_j^i \in R_i$ **do**
 - 3 **foreach** each attribute $o_j^i.x_k$ **do**
 - 4 set $E(s_k) \leftarrow \langle \text{EHL}(o_j^i.x_k), \text{Enc}(o_j^i.x_k) \rangle$;
 - 5 set $E(o_j^i) = (E(s_1), \dots, E(s_{m_i}))$
 - 6 Generate a key K for the PRP P ;
 - 7 Permutes the encrypted attributes based on P , i.e. set $E(o_j^i) = (E(s_{P_K(1)}), \dots, E(s_{P_K(m_i)}))$;
 - 8 Output permuted encrypted databases as $\text{ER}_1 = \{E(o_1^1) \dots E(o_{n_1}^1)\}$ and $\text{ER}_2 = \{E(o_1^2) \dots E(o_{n_2}^2)\}$;
-

Consider a set of relations R_1 and R_2 . The encryption setup is similar as the top- k for one relation. The difference is that since we have multiple relations on different data we cannot assign a global object identifier for each the objects in different relations. The difference here is that, in addition to encrypting an object id with EHL, we encrypt the attribute value using EHL since the join condition generated from the client is to join the relations based on the attribute values. Therefore, we can compare the equality between different records based on their attributes. The encryption $\text{Enc}(R_1, R_2)$ is given in Algorithm 10.

The encrypted relations ER_1, ER_2 do not reveal anything besides the size. The proof is similar to the proof in Theorem 6.1.

12.3 Query Token

Consider a client that wants to run query a SQL-like top- k join as follows: $Q = \text{SELECT } * \text{ FROM } R_1, R_2 \text{ WHERE } R_1.A = R_2.B \text{ ORDER BY } R_1.C + R_1.D \text{ STOP AFTER } k$; where A, C are attributes in R_1 and B, D are attributes in R_2 . The client first requests the key K for the P , then computes $(t_1, t_2, t_3, t_4) \leftarrow (P_K(R_1.A), P_K(R_2.B), P_K(R_1.C), P_K(R_2.D))$. Finally, the client generates the SQL-like query token as follows: $t_Q = \text{SELECT } * \text{ FROM } \text{ER}_1, \text{ER}_2 \text{ WHERE } \text{ER}_1.t_1 = \text{ER}_2.t_2 \text{ ORDERED BY } \text{ER}_1.t_3 + \text{ER}_2.t_4 \text{ STOP AFTER } k$. Then, the client sends the token t_Q to the server S_1 .

12.4 Query Processing for top-k join

In this section, we introduce the secure top- k join operator \bowtie_{sec} . We first introduce some notation that we use in the query processing algorithm. For a receiving token t_Q that is described in Section 12.4, let the join condition be $\text{JC} \stackrel{\text{def}}{=} (\text{ER}_1.t_1 = \text{ER}_2.t_2)$, and the score function $\text{Score} = \text{ER}_1.t_3 + \text{ER}_2.t_4$. Moreover, for each $E(o_i^1) \in \text{ER}_1$, let $E(x_{it_1})$ and $E(x_{it_3})$ be the t_1 -th and t_3 -th encrypted attribute. Similarly, let $E(x_{jt_2})$ and $E(x_{jt_4})$ be the t_2 -th and t_4 -th encrypted attribute for each $E(o_j^2)$ in ER_2 . In addition, let $\mathbf{E}(X)$ be a vector of encryptions, i.e. $\mathbf{E}(X) = \langle \text{Enc}(x_1), \dots, \text{Enc}(x_s) \rangle$, and let $\mathbf{E}(R) = \langle (\text{Enc}(r_1), \dots, \text{Enc}(r_s)) \rangle$, where $R \in \mathbb{Z}_N^s$ with each $r_i \xleftarrow{\$} \mathbb{Z}_N$. Denote the randomization function Rand

Algorithm 11: $\text{SecJoin}(\text{tk}, \text{pk}_p, \text{sk}_p)$: \bowtie_{sec} with $\text{JC} = (\text{ER}_1.t_1, \text{ER}_2.t_2)$ and $\text{Score} = \text{ER}_1.t_3 + \text{ER}_2.t_4$

S_1 's input: pk_p, tk
 S_2 's input: pk_p, sk_p

- 1 **Server S_1 :**
- 2 Parse tk , let $\text{JC} = (\text{ER}_1.t_1, \text{ER}_2.t_2)$ and $\text{Score} = \text{ER}_1.t_3 + \text{ER}_2.t_4$
- 3 **foreach** $E(o_i^1) \in \text{ER}_1, E(o_j^2) \in \text{ER}_2$ *in random order* **do**
- 4 Let $E(o_i^1) = (E(x_{i1}), \dots, E(x_{im_1}))$ and $E(o_j^2) = (E(x_{j1}), \dots, E(x_{jm_2}))$;
- 5 Compute $\text{Enc}(b_{ij}) \leftarrow \text{EHL}(x_{it_1}) \ominus \text{EHL}(x_{jt_2})$, where $E(x_{it_1}), E(x_{jt_2})$ are the t_1 -th, t_2 -th attributes in $E(o_i^1), E(o_j^2)$;
- 6 /* evaluate $s_{ij} = b_{ij}(x_{it_3} + x_{jt_4})$ */
- 7 Send $\text{Enc}(b_{ij})$ to S_2 and receive $\mathcal{E}^2(t_{ij})$ from S_2 ;
- 8 Compute Score as: $s_{ij} \leftarrow \mathcal{E}^2(b_{ij})^{\text{Enc}(x_{it_3})\text{Enc}(x_{jt_4})}$;
- 9 run $\text{Enc}(s_{ij}) \leftarrow \text{RecoverEnc}(S_{ij}, \text{pk}_p, \text{sk}_p)$.
- 10 Combine rest of the attributes for $E(o_{ij})$ as follows: $x_l \leftarrow \mathcal{E}^2(b_{ij})^{\text{Enc}(x_l)}$, where $\text{Enc}(x_l) \in \{\text{Enc}(x_{i1}) \dots \text{Enc}(x_{jm_2})\}$ in $E(o_i^1), E(o_j^2)$;
- 11 Run $\text{Enc}(x_l) \leftarrow \text{RecoverEnc}(x_l, \text{pk}_p, \text{sk}_p)$.
- 12 **Server S_2 :**
- 13 For each received t_{ij} , decrypts it. If it is 0, then compute $b_{ij} \leftarrow \mathcal{E}^2(1)$. Otherwise, $b_{ij} \leftarrow \mathcal{E}^2(0)$. Sends b_{ij} to S_1 .
- 14 **Server S_1 :**
- 15 Finally holds joined encrypted tuples $E(o_{ij}) = \text{Enc}(s_{ij}), \{\text{Enc}(x_l)\}$, where $\text{Enc}(s_{ij})$ is the encrypted Score , $\{\text{Enc}(x_l)\}$ are the joined attributes from ER_1, ER_2 .
- 16 Run $L \leftarrow \text{SecFilter}(\{E(o_{ij}), \text{pk}_p, \text{sk}_p\})$ and get the encrypted list L .
- 17 Run EncSort to conduct encrypted sort on encrypted Score $\text{Enc}(s_{ij})$, and return top- k encrypted items.

as below:

$$\begin{aligned} \text{Rand}(\mathbf{E}(X), \mathbf{E}(R)) &= (\text{Enc}(x_1) \cdot \text{Enc}(r_1), \dots, \text{Enc}(x_n) \cdot \text{Enc}(r_n)) \\ &= (\text{Enc}(x_1 + r_1), \dots, \text{Enc}(x_n + r_n)) \end{aligned}$$

. This function is similar to Rand in Algorithm 7 and is used to homomorphically blind the original value.

In general, the procedure for query processing includes the following steps:

- Perform the join on ER_1 and ER_2 .
 - Receiving the token, S_1 runs the protocol with S_2 to generate all possible joined tuples from two relations and homomorphically computes the encrypted scores.
 - After getting all the joined tuples, S_1 runs $\text{SecFilter}(\{E(o_i)\}, \text{pk}_p, \text{sk}_p)$ (see Algorithm 12), which is a protocol with S_2 to eliminate the tuples that do not meet the join condition. S_1 and S_2 then runs the protocol SecJoin . S_1 finally produce the encrypted join tuples together with their scores.
- EncSort : after securely joining all the databases, S_1 then runs the encrypted sorting protocol to get the top- k results.

The main \bowtie_{sec} is fully described in Algorithm 11. As mentioned earlier, since all the attributes are encrypted, we cannot simply use the traditional join strategy. The merge-sort or hash based join

Algorithm 12: SecFilter($\{E(o_i)\}, \text{pk}_p, \text{sk}_p$)

S_1 's input: $\{E(o_i)\}, \text{pk}_p$
 S_2 's input: pk_p, sk_p

```

1  Server  $S_1$ :
2  | Let  $E(o_i) = (\text{Enc}(s_i), \mathbf{E}(X_i))$  where  $\mathbf{E}(X_i) = \langle \text{Enc}(x_{i1}), \dots, \text{Enc}(x_{is}) \rangle$ ;
3  | Generate a key pair  $(\text{pk}_s, \text{sk}_s)$ ;
4  | foreach  $E(o_i)$  do
5  | | Generate random  $r_i \in \mathbb{Z}_N^*$ , and  $R_i \in \mathbb{Z}_N^m$ ;
6  | |  $\text{Enc}(s'_i) \leftarrow \text{Enc}(s_i)^{r_i}$  and  $\mathbf{E}(X'_i) \leftarrow \text{Rand}(\mathbf{E}(X_i), \mathbf{E}(R_i))$ ;
7  | | Set  $E(o'_i) = (\text{Enc}(s'_i), \mathbf{E}(X'_i))$ ;
8  | Compute the following:  $\text{Enc}_{\text{pk}_s}(r_1^{-1}), \text{Enc}_{\text{pk}_s}(R_1), \dots, \text{Enc}_{\text{pk}_s}(r_n^{-1}), \text{Enc}_{\text{pk}_s}(R_n)$ ;
9  | Generate random permutation  $\pi$ , permute  $E(o'_{\pi(i)}), \text{Enc}_{\text{pk}_s}(r_{\pi(i)}^{-1})$ , and  $\text{Enc}_{\text{pk}_s}(R_{\pi(i)})$ ;
10 | Sends  $E(o'_{\pi(i)}), \text{Enc}_{\text{pk}_s}(r_{\pi(i)}^{-1}), \text{Enc}_{\text{pk}_s}(R_{\pi(i)})$ , and  $\text{pk}_s$  to  $S_2$ ;

11 Server  $S_2$ :
12 | Receiving the list  $E(o'_{\pi(i)}), \text{Enc}_{\text{pk}_s}(r_{\pi(i)}^{-1}), \text{Enc}_{\text{pk}_s}(R_{\pi(i)})$ ;
13 | foreach  $E(o'_{\pi(i)}) \in (\text{Enc}(s'_{\pi(i)}), \mathbf{E}(X'_{\pi(i)}))$  do
14 | | decrypt  $b \leftarrow \text{Enc}(s'_{\pi(i)})$ ;
15 | | if  $b = 0$  then
16 | | | Remove this entry  $E(o'_{\pi(i)}), \text{Enc}_{\text{pk}_s}(r_{\pi(i)}^{-1}), \text{Enc}_{\text{pk}_s}(R_{\pi(i)})$ 
17 | foreach remaining items do
18 | | Generate random  $\gamma_i \in \mathbb{Z}_N^*$ , and  $\Gamma_i \in \mathbb{Z}_N^m$ ;
19 | |  $\text{Enc}(\tilde{s}_i) \leftarrow \text{Enc}(s'_{\pi(i)})^{\gamma_i}$  and  $\mathbf{E}(\tilde{X}_i) \leftarrow \text{Rand}(\mathbf{E}(X'_{\pi(i)}), \mathbf{E}(\Gamma_i))$ ;
20 | | Set  $E(\tilde{o}_i) = (\text{Enc}(\tilde{s}_i), \mathbf{E}(\tilde{X}_i))$ ;
21 | | /* evaluate  $\tilde{r}_i = r_{\pi(i)}^{-1} \gamma_i^{-1}$  */ /*
22 | | compute the following using  $\text{pk}_s$ :  $\text{Enc}_{\text{pk}_s}(\tilde{r}_i) \leftarrow \text{Enc}_{\text{pk}_s}(r_{\pi(i)}^{-1})^{\gamma_i^{-1}}$ ;
23 | | /* evaluate  $\tilde{R}_i = R_{\pi(i)} + \Gamma_i^{-1}$  */ /*
24 | |  $\text{Enc}_{\text{pk}_s}(\tilde{R}_i) \leftarrow \text{Enc}_{\text{pk}_s}(R_{\pi(i)}) \cdot \text{Enc}_{\text{pk}_s}(\Gamma_i)$ 
25 | | Sends the  $E(\tilde{o}_i)$  and  $\text{Enc}_{\text{pk}_s}(\tilde{r}_i), \text{Enc}_{\text{pk}_s}(\tilde{R}_i)$  to  $S_1$ 

26 Server  $S_1$ :
27 | foreach  $E(\tilde{o}_i) = (\text{Enc}(\tilde{s}_i), \mathbf{E}(\tilde{X}_i))$  and  $\text{Enc}_{\text{pk}_s}(\tilde{r}_i), \text{Enc}_{\text{pk}_s}(\tilde{R}_i)$  do
28 | | use  $\text{sk}_s$  to decrypt  $\text{Enc}(\tilde{r}_i)$  as  $\tilde{r}_i$ ,  $\text{Enc}(\tilde{R}_i)$  as  $\tilde{R}_i$ ;
29 | | /* homomorphically de-blind */
30 | | compute  $\text{Enc}(\bar{s}_i) \leftarrow \text{Enc}(\tilde{s}_i)^{\tilde{r}_i}$  and  $\mathbf{E}(\bar{X}_i) \leftarrow \text{Rand}(\mathbf{E}(\tilde{X}_i), \mathbf{E}(-\tilde{R}_i))$ ;
31 | | Set  $E(o'_i) = (\text{Enc}(\bar{s}_i), \mathbf{E}(\bar{X}_i))$ ;
32 | | /* Suppose there're  $l$  tuples left */ /*
33 | Output the list  $E(o'_1) \dots E(o'_l)$ .

```

cannot be applied here since all the tuples have been encrypted by a probabilistic encryption. Our idea for S_1 to securely produce the joined result is as follows: S_1 first combines all the tuples from two databases (say, using nested loop) by initiating the protocol **SecJoin**. After that, S_1 holds all the combined tuples together with the scores. The joined tuple have $m_1 + m_2$ many attributes (or user selected attributes). Those tuples that meet the equi-join condition **JC** are successfully joined together with the encrypted scores that satisfy the **Score** function. However, for those tuples that do not meet the **JC**, their encrypted scores are homomorphically computed as $\text{Enc}(0)$ and their joined attributes are all $\text{Enc}(0)$ as well. S_1 holds all the possible combined tuples. Next, the **SecFilter** eliminates all of those tuples that do not satisfy **JC**. It is easy to see that similar techniques from **SecDupElim** can be applied here. At the end of the protocol, both S_1 and S_2 only learn the final number of the joined tuples that

meet JC.

Below we describe the SecJoin and SecFilter protocols in detail. Receiving the Token, S_1 first parses it as the join condition $JC = (ER_1.t_1, ER_2.t_2)$, and the score function $Score = ER_1.t_3 + ER_2.t_4$. Then for each encrypted objects $E(o_{1i}) \in ER_1$ and $E(o_{2i}) \in ER_2$ in random order S_1 computes $t_{ij} \leftarrow (EHL(x_{it_1}) \ominus EHL(x_{jt_2}))^{r_{ij}}$, where x_{it_1} and x_{jt_2} are the value for the t_1 th and t_2 th attribute for $E(o_{1i})$ and $E(o_{2j})$ separately. r_{ij} is randomly generated value in \mathbb{Z}_N^* , then S_1 sends t_{ij} to S_2 . Having the decryption key, S_2 decrypts it to b_{ij} , which indicates whether the encrypted value x_{it_1} and x_{jt_2} are equal or not. If $b_{ij} = 0$, then we have $x_{it_1} = x_{jt_2}$ which meets the join condition JC. Otherwise, b_{ij} is a random value. S_2 then encrypts the bit b_{ij} using a double layered encryption and sends it to S_1 , where $b_{ij} = 0$ if $x_{it_1} \neq x_{jt_2}$ otherwise $b_{ij} = 1$. Receiving the encryption, S_1 computes $s_{ij} \leftarrow \mathcal{E}^2(b_{ij})^{\text{Enc}(x_{1t_3})\text{Enc}(x_{2t_4})}$, where x_{1t_3} is the t_3 -th attribute for $E(o_{1i})$ and x_{2t_4} is the t_4 -th attribute for $E(o_{2j})$. Finally, S_1 runs the StripEnc to get the normal encryption $\text{Enc}(s_{ij})$. Based on the construction,

$$\text{Enc}(s_{ij}) \sim \text{Enc}(b_{ij}(x_{1t_3} + x_{2t_4})), \text{ where } b_{ij} = \begin{cases} 1 & \text{if } x_{1t_1} = x_{2t_2} \\ 0 & \text{otherwise} \end{cases}$$

Finally, after fully combining the encrypted tuples, S_1 holds the joined encrypted tuple as well as the encrypted scores, i.e. $E(T) = (\text{Enc}(s_{ij}), \text{Enc}(x_{11})\dots\text{Enc}(x_{1m_1}), \text{Enc}(x_{21}), \dots, \text{Enc}(x_{2m_2}))$. During the execution above, nothing has been revealed to S_1 , S_2 only learns the number of tuples meets the join condition JC but does not which pairs since the S_1 sends out the encrypted values in random order. Also, notice that S_1 can only select interested attributes from ER_1 and ER_2 when combining the encrypted tuples. Here we describe the protocol in general.

After SecJoin, assume S_1 holds n combined the tuples with each tuple has m combined attributes, then for each of tuple $E(T_i) = (\text{Enc}(s_i), \mathbf{E}(X_i))$, where $\mathbf{E}(X_i)$ is the combined encrypted attributes $\mathbf{E}(X_i) = \langle \text{Enc}(x_{i1}), \dots, \text{Enc}(x_{im}) \rangle$. Next, S_1 tries to blind encryptions in order to prevent S_2 from knowing the actual value. For each $E(T_i)$, S_1 generates random $r_i \in \mathbb{Z}_N^*$ and $R_i \in \mathbb{Z}_N^m$, and blinds the encryption by computing following: $\text{Enc}(s'_i) \leftarrow \text{Enc}(s_i)^{r_i}$ and $\mathbf{E}(X'_i) \leftarrow \text{Rand}(\mathbf{E}(X_i), \mathbf{E}(R_i))$. Then S_1 sets $E(T'_i) = (\text{Enc}(s'_i), \text{Enc}(X'_i))$. Furthermore, S_1 generates a new key pair for the paillier encryption scheme (pk_s, sk_s) and encrypts the following: $L = \text{Enc}(r_1^{-1})_{pk_s}, \text{Enc}(R_1)_{pk_s}, \dots, \text{Enc}(r_n^{-1})_{pk_s}, \text{Enc}(R_n)_{pk_s}$, where each r_i^{-1} is the inverse of r_i in the group \mathbb{Z}_N . S_1 needs to encrypt the randomnesses in order to recover the original values, and we will explain this later. Moreover, S_1 generates a random permutation π , then permutes $E(T_{\pi(i)})$ and $\text{Enc}(r_{\pi(i)}^{-1})_{pk_s}, \text{Enc}(R_{\pi(i)})_{pk_s}$ for $i = [1, n]$. S_1 sends the permuted encryptions to S_2 .

S_2 receives all the encryptions. For each received $E(T'_{\pi(i)}) = (\text{Enc}(s'_{\pi(i)}), \mathbf{E}(X'_{\pi(i)}))$, S_2 decrypts $\text{Enc}(s'_{\pi(i)})$, if $s'_{\pi(i)}$ is 0 then S_2 removes tuple $E(T'_{\pi(i)})$ and corresponding $\text{Enc}(r_{\pi(i)}^{-1})_{pk_s}, \text{Enc}(R_{\pi(i)})_{pk_s}$. For the remaining tuples $E(T_{\pi(i)})$, S_2 generates random $r'_i \in \mathbb{Z}_N^*$, and $R'_i \in \mathbb{Z}_N^m$, and compute the following $\text{Enc}(\tilde{s}_i) \leftarrow \text{Enc}(s'_{\pi(i)})^{r'_i}$, $\text{Enc}(\tilde{X}_i) \leftarrow \text{Rand}(\mathbf{E}(X'_{\pi(i)}), \mathbf{E}(R'_i))$ (see Algorithm 11 line 19). Then set $E(\tilde{T}_i) = (\text{Enc}(\tilde{s}_i), \text{Enc}(\tilde{X}_i))$. Note that, this step prevents the S_1 from knowing which tuples have been removed. Also, in order to let S_1 recover the original values, S_2 encrypts and compute the following using pk_s , $\text{Enc}(\tilde{r}_i) \leftarrow \text{Enc}(r_{\pi(i)}^{-1})^{r'_i}_{pk_s}$ and $\text{Enc}(\tilde{R}_i) \leftarrow \text{Enc}(R'_{\pi(i)})_{pk_s} \cdot \text{Enc}(R'_i)_{pk_s}$. Finally, S_1 sends the $E(\tilde{T}_i)$ and $\text{Enc}(\tilde{r}_i), \text{Enc}(\tilde{R}_i)$ to S_1 . Assuming there're n' joined tuples left. On the other side, S_1 receives the encrypted tuples, for each $E(\tilde{T}_i) = \text{Enc}(\tilde{s}), \text{Enc}(\tilde{X}_i)$ and $\text{Enc}(\tilde{r}_i), \text{Enc}(\tilde{R}_i)$, S_1 recovers the original values by computing the following: compute $\text{Enc}(s'_i) \leftarrow \text{Enc}(\tilde{s}_i)^{\tilde{r}_i}$ and $\mathbf{E}(X'_i) \leftarrow \text{Rand}(\mathbf{E}(\tilde{X}_i), \mathbf{E}(-\tilde{R}_i))$. Notice that, for the remaining encrypted tuples and their encrypted scores, S_1 can successfully recover the original value, we show below that the encrypted scores $\text{Enc}(\tilde{s}_j)$ is indeed some permuted $\text{Enc}(s_{\pi(i)})$:

$$\begin{aligned}
\text{Enc}(\bar{s}_j) &\sim \text{Enc}(\tilde{r}_j \cdot \tilde{s}_j) && \text{(see Alg. 11 line 28)} \\
&\sim \text{Enc}(r_{\pi(j)}^{-1} r_j'^{-1} \cdot \tilde{s}_j) && \text{(see Alg. 11 line 22)} \\
&\sim \text{Enc}(r_{\pi(j)}^{-1} r_j'^{-1} \cdot s_{\pi(j)}' \cdot r_j') && \text{(see Alg. 11 line 19)} \\
&\sim \text{Enc}(r_{\pi(j)}^{-1} r_j'^{-1} \cdot s_{\pi(j)} r_{\pi(j)} r_j') && \text{(see Alg. 11 line 6,10)} \\
&\sim \text{Enc}(s_{\pi(j)})
\end{aligned}$$

If we don't want to leak the number of tuples that meet , we can use a similar technique from SecDedup, that is, S_2 generates some random tuples and large enough random scores for the tuples to not satisfy JC. In this way, nothing else has been leaked to the servers. It is worth noting that the technique sketched above not only can be used for top- k join, but for any equality join can be applied here.

12.4.1 Performance Evaluation

We conduct the experiments under the same environment as in Section 11. We use synthetic datasets to evaluate our sec-join operator \bowtie_{sec} : we uniformly generate R_1 with 5K tuples and 10 attributes, and R_2 with 10K tuples and 15 attributes. Since the server runs the *oblivious join* that we discuss before over the encrypted databases, the performance of the \bowtie_{sec} does not depend on the parameter k . We test the effect of the joined attributes in the experiments. We vary the total number of the attribute m joined together from two tables. Figure 13 shows performance when m ranges from 5 to 20.

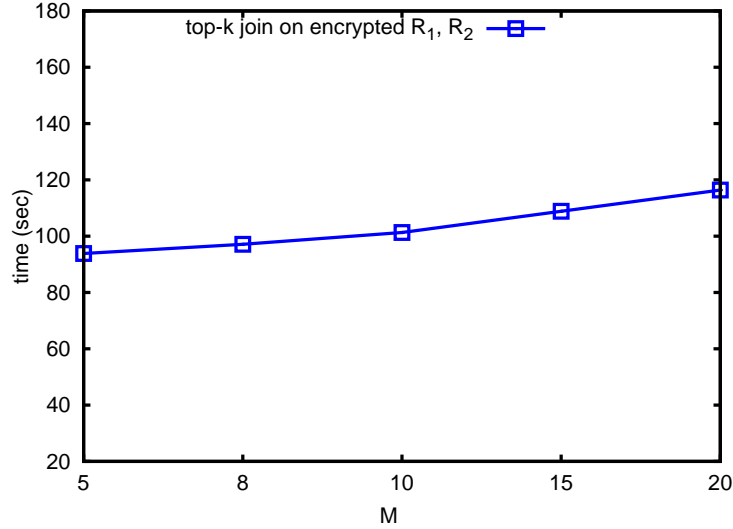


Figure 13: Top- k join: \bowtie_{sec}

Our operator \bowtie_{sec} is generically designed for joining any attributes between two relations. In practice, one would be only interested in joining two tables using primary-key-to-foreign-key join or foreign-key-to-primary-key join. Our methods can be easily generalized to those joins. In addition, one can also pre-sort the attributes to be ranked and save computations in the \bowtie_{sec} processing. We leave this as the future work of this thesis.

12.5 Related works on Secure Join

Many works have proposed for executing equi-joins over encrypted data. One recent work [39] proposed a privacy-preserving join on encrypted data. Their work mainly designed for the private join

operation, therefore cannot support the top- k join. In addition, in [39], although the actual values for the joined records are not revealed, the server learns some equality pattern on the attributes if records are successfully joined. In addition, [39] uses bilinear pairing during their query processing, thus it might cause high computation overhead for large datasets. CryptDB [40] is a well-known system for processing queries on encrypted data. MONOMI [45] is based on CryptDB with a special focus on efficient analytical query processing. [31] adapts the deterministic proxy re-encryption to provide the data confidentiality. The approaches using deterministic encryption directly leak the duplicates and, as a result, the equality information are leaked to the adversarial servers. [49] propose a secure query system SDB that protects the data confidentiality by decomposing the sensitive data into shares and can perform secure joins on shares of the data. However, it is unclear whether the system can perform top- k queries over the shares of the data. Other solutions such as Order-preserving encryption (OPE) [10, 4] can also be adapted to secure top- k join, however, it is commonly not considered very secure on protecting the ranks of the score as the adversarial server directly learns the order of the attributes.

13 Top- k Query Processing Conclusion

This paper proposes the first complete scheme that executes top- k ranking queries over encrypted databases in the cloud. First, we describe a secure probabilistic data structure called encrypted hash list (EHL) that allows a cloud server to homomorphically check equality between two objects without learning anything about the original objects. Then, by adapting the well-known NRA algorithm, we propose a number of secure protocols that allow efficient top- k queries execution over encrypted data. The protocols proposed can securely compute the best/worst ranking scores and de-duplication of the replicated objects. Moreover, the protocols in this paper are stand-alone which means the protocols can be used for other applications besides the secure top- k query problem. We also provide a clean and formal security analysis of our proposed methods where we explicitly state the leakage of various schemes. The scheme has been formally proved to be secure under the CQA security definition and it is experimentally evaluated using real-world datasets which show the scheme is efficient and practical.

References

- [1] B. Adida and D. Wikström. How to shuffle in public. In *TCC*, pages 555–574, 2007.
- [2] C. C. Aggarwal and P. S. Yu, editors. *Privacy-Preserving Data Mining-Models and Algorithms*, volume 34 of *Advances in Database Systems*. 2008.
- [3] D. Agrawal, A. El Abbadi, F. Emekci, A. Metwally, and S. Wang. Secure data management service on cloud computing infrastructures. In *New Frontiers in Information and Software as Services*, volume 74, pages 57–80. 2011.
- [4] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *ACM SIGMOD*, pages 563–574, 2004.
- [5] A. Arasu, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, and R. Ramamurthy. Transaction processing on confidential data using cipherbase. In *ICDE*, pages 435–446, 2015.
- [6] S. Bajaj and R. Sion. Trusteddb: a trusted hardware based database with privacy and data confidentiality. In *SIGMOD*, pages 205–216, 2011.
- [7] F. Baldimtsi and O. Ohrimenko. Sorting and searching behind the curtain. In *FC*, 2014.

- [8] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. *PVLDB*, 9(7):528–539, 2016.
- [9] A. Boldyreva, N. Chenette, and A. O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, pages 578–595, 2011.
- [10] A. Boldyreva, N. Chenette, and A. O'Neill. Order-preserving encryption revisited: improved security analysis and alternative solutions. In *CRYPTO '11*, pages 578–595, 2011.
- [11] R. Bost, R. A. Popa, S. Tu, and S. Goldwasser. Machine learning classification over encrypted data. In *NDSS*, 2015.
- [12] S. Bugiel, S. Nürnberger, A. Sadeghi, and T. Schneider. Twin clouds: Secure cloud computing with low latency. In *CMS*, pages 32–44, 2011.
- [13] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO*, 2013.
- [14] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *ASIACRYPT*, pages 577–594, 2010.
- [15] S. Choi, G. Ghinita, H. Lim, and E. Bertino. Secure knn query processing in untrusted cloud environments. *TKDE*, 26:2818–2831, 2014.
- [16] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *CCS 2006*, pages 79–88. ACM, 2006.
- [17] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. *Journal of Computer Security*, pages 895–934, 2011.
- [18] I. Damgård and M. Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *PKC*, pages 119–136, 2001.
- [19] D. Demmler, T. Schneider, and M. Zohner. ABYA framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.
- [20] C. Dwork and K. Nissim. Privacy-preserving datamining on vertically partitioned databases. In *CRYPTO*, pages 528–544, 2004.
- [21] Y. Elmehdwi, B. K. Samanthula, and W. Jiang. Secure k-nearest neighbor query over encrypted data in outsourced environments. In *ICDE*, pages 664–675, 2014.
- [22] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.
- [23] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.
- [24] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43:431–473, 1996.
- [25] H. Hacigümüs, B. R. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *SIGMOD*, pages 216–227, 2002.

- [26] I. Hang, F. Kerschbaum, and E. Damiani. ENKI: access control for encrypted query processing. In *SIGMOD*, pages 183–196, 2015.
- [27] B. Hore, S. Mehrotra, M. Canim, and M. Kantarcioglu. Secure multidimensional range queries over outsourced data. *VLDB J.*, 21(3):333–358, 2012.
- [28] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top- k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.
- [29] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.
- [30] C. C. Jaideep Vaidya, Murat Kantarcioglu. Privacy-preserving naïve bayes classification. *VLDB J.*, 17(4):879–898, 2008.
- [31] F. Kerschbaum, M. Härterich, P. Grofig, M. Kohler, A. Schaad, A. Schröpfer, and W. Tighzert. Optimal re-encryption strategy for joins in encrypted databases. In *DBSec*, pages 195–210, 2013.
- [32] M. Kuzu, M. S. Islam, and M. Kantarcioglu. Efficient privacy-aware search over encrypted databases. *CODASPY*, pages 249–256, 2014.
- [33] R. Li, A. X. Liu, A. L. Wang, and B. Bruhadeshwar. Fast range query processing with strong privacy protection for cloud computing. *PVLDB*, 7(14):1953–1964, 2014.
- [34] M. Lichman. UCI machine learning repository, 2013.
- [35] Y. Lindell and B. Pinkas. Privacy preserving data mining. In *CRYPTO*, pages 36–54, 2000.
- [36] A. Liu, K. Zheng, L. Li, G. Liu, L. Zhao, and X. Zhou. Efficient secure similarity computation on encrypted trajectory data. In *ICDE*, pages 66–77, 2015.
- [37] C. C. Murat Kantarcioglu. Privacy-preserving distributed mining of association rules on horizontally partitioned data. *TKDE*, 16:1026–1037, 2004.
- [38] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, pages 223–238, 1999.
- [39] H. Pang and X. Ding. Privacy-preserving ad-hoc equi-join on outsourced data. *ACM Trans. Database Syst.*
- [40] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *SOSP*, pages 85–100, 2011.
- [41] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas. Constants count: Practical improvements to oblivious ram. In *USENIX Security*, 2015.
- [42] B. K. Samanthula, W. Jiang, and E. Bertino. Privacy-preserving complex query evaluation over semantically secure encrypted data. In *ESORICS*, pages 400–418, 2014.
- [43] E. Shi, J. Bethencourt, H. T. Chan, D. X. Song, and A. Perrig. Multi-dimensional range query over encrypted data. In *SECP*, pages 350–364, 2007.
- [44] D. Song, D. Wagner, and A. Perrig. Practical techniques for searching on encrypted data. In *Oakland S & P*, pages 44–55, 2000.
- [45] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. *PVLDB*, 6(5):289–300, 2013.

- [46] J. Vaidya and C. Clifton. Privacy-preserving top-k queries. In *ICDE*, pages 545–546, 2005.
- [47] J. Vaidya, C. Clifton, M. Kantarcioglu, and A. S. Patterson. Privacy-preserving decision trees over vertically partitioned data. *TKDD*, 2(3), 2008.
- [48] W. K. Wong, D. W.-l. Cheung, B. Kao, and N. Mamoulis. Secure knn computation on encrypted databases. In *SIGMOD*, pages 139–152, 2009.
- [49] W. K. Wong, B. Kao, D. W. Cheung, R. Li, and S. Yiu. Secure query processing with data interoperability in a cloud database environment. In *SIGMOD*, pages 1395–1406, 2014.
- [50] B. Yao, F. Li, and X. Xiao. Secure nearest neighbor revisited. In *ICDE*, pages 733–744, 2013.

Appendix A Security between S_1 and S_2

Informally, our privacy definitions capture the following: S_1 learns the number of documents and the halting depth, as well as the query pattern and access pattern of client queries since he observes the “encrypted” queries of the client. However, S_1 learns nothing about the objects and scores as they are all protected by the semantic encryption scheme. S_2 , on the other hand, only learns the equality pattern at each depth when executing the protocol *SecQuery* with S_1 . In other words, the cryptography cloud S_2 learns some pattern from the encrypted data when the client is querying the system. However, S_2 learns nothing about the access and query patterns, and hence does not know anything about how does S_1 access the encrypted relation, nor the content of the queries. We note that S_1 ’s capabilities are similar to those of the server in the original searchable encryption and structured encryption definition [16, 14] while S_2 learns much less.

We formalize the privacy between the semi-honest adversaries S_1 and S_2 as follows: Let Π_f be a two-party protocol that supports the functionality f during the query processing. Let $\text{View}_{S_i}^{\Pi_f}(I)$ be all the transcripts that S_i receives while running the protocol on input I , and let Sim_{S_i} be the simulator who tries to simulate the views for S_i . While executing the protocol, let $\text{Sim}_{S_i}(\mathcal{L}_{S_i}(\cdot))$ denote the simulated transcript by Sim_{S_i} on the input $\mathcal{L}_{S_i}(\cdot)$, where $\mathcal{L}_{S_i}(\cdot)$ is a stateful function that Sim_{S_i} learns during the execution of the protocol. At the beginning, S_1 has the encrypted database *ER* and the public key *pk*, while S_2 holds the public/secret key *pk* and *sk*.

Definition A.1. Let Π_f be a two party protocol for computing a function f . S_1 takes the input (pk_p, x) and S_2 takes the input $(\text{pk}_p, \text{sk}_p, y)$. When Π_f terminates S_1 receives $f(x, y)$. Let $\text{View}_{S_i}^{\Pi_f}(\text{pk}_p, \text{sk}_p, x, y)$ be all the message that S_i receives while running the protocol on input $(\text{pk}_p, \text{sk}_p, x, y)$ and Output^f be the output of the protocol received by S_1 . We say that the protocol Π_f between S_1 and S_2 privately realizes the functionality f if there exists a pair of probabilistic polynomial time (PPT) simulators Sim_{S_1} and Sim_{S_2} such that:

$$\begin{aligned}
 (1). & \left(\text{Sim}_{S_2}(\text{pk}_p, \mathcal{L}_{S_2}), \Pi_f(\text{pk}_p, \text{sk}_p, x, y) \right) \cong \left(\text{View}_{S_1}^{\Pi_f}(\text{pk}_p, \text{sk}_p, x, y), \text{Output}^f(\text{pk}_p, \text{sk}_p, x, y) \right) \\
 (2). & \left(\text{Sim}_{S_1}(\text{pk}_p, \text{sk}_p, \mathcal{L}_{S_1}) \cong \text{View}_{S_2}^{\Pi_f}(\text{pk}_p, \text{sk}_p, x, y) \right)
 \end{aligned}$$

The two sub-routines *EncSort* and *EncCompare* have been proved to be secure in [7, 11] and we refer the proofs from there. Below we show that *SecWorst* we describe in the section 8.2.1 is secure based on the security definition A.1. In addition, the security of *SecBest*, *SecDedup*, *SecUpdate* are very similar to the proof of *SecWorst*.

Lemma A.2. Let SecWorst be the protocol between S_1 and S_2 described in Algorithm 4 that privately computes the SecWorst functionality at depth d . As describe in Protocol 8.1 S_1 takes as input $(\text{pk}_p, E(I), H)$ and S_2 takes as input $(\text{pk}_p, \text{sk}_p)$. When SecWorst terminates S_1 receives the output $\text{Enc}(W)$ such that W is the worst score based on the list H , and let $\mathcal{L}_{s_1} = (|H|, \text{EP}^d(\text{ER}, q))$, $\mathcal{L}_{s_2} = |\text{ER}|$. Then Π_{SecWorst} is secure based on the definition A.1, i.e.,

$$\begin{aligned} (1). & \left(\text{Sim}_{s_2}(\text{pk}_p, \mathcal{L}_{s_2}), \text{SecWorst}(\text{pk}_p, \text{sk}_p, E(I), H) \right) \cong \\ & \left(\text{View}_{S_1}^{\text{SecWorst}}(\text{pk}_p, \text{sk}_p, E(I), H), \text{Output}^{\text{SecWorst}}(\text{pk}_p, \text{sk}_p, E(I), H) \right) \\ (2). & \left(\text{Sim}_{s_1}(\text{pk}_p, \text{sk}_p, \mathcal{L}_{s_1}) \cong \text{View}_{S_2}^{\text{SecWorst}}(\text{pk}_p, \text{sk}_p, E(I), H) \right) \end{aligned}$$

Proof: We construct Sim_{s_2} as follows. Sim_{s_2} behaves like S_1 and interacts with S_2 . On the input of the list H and the encrypted item $E(I)$, Sim_{s_2} runs the SecWorst using the homomorphic properties from EHL and encryptions and, hence, learns nothing more than what can be already computed from the inputs. Sim_{s_2} simply just follows the SecWorst protocol. First, Sim_{s_2} generates the random permutation π , then sends the encrypted $\text{Enc}(b_j)$ to S_2 based on the description of the SecWorst protocol. When receiving the encrypted bit $\mathcal{E}^2(t_i)$, Sim_{s_2} computes the ciphertexts specified in SecWorst . Sim_{s_2} continues by homomorphically computing the $\prod_{i=1}^S \text{Enc}(x'_i)$. Finally, based on the protocol SecWorst , Sim_{s_2} obtains encrypted worst score $\text{Enc}(W)$. Since all of the messages are encrypted under semantically secure encryption. Sim_{s_2} learns nothing from the protocol SecWorst . On the other hand, Sim_{s_1} , similar as S_2 , has the access to the keys pk_p, sk_p and, hence, can decrypt the message received from S_1 . Sim_{s_1} also gets the $\mathcal{L}_{s_1} = (|H|, \text{EP}^d(\text{ER}, q))$ at depth d for some query q . By receiving $\text{Enc}(b_i)$ from S_1 and gets \mathcal{L}_{s_1} , Sim_{s_1} can simulates the views for S_1 and behaves exactly the same as S_2 . As described in SecWorst , Sim_{s_1} sends the $\mathcal{E}^2(t_i)$ based on the output of L_{s_1} . Therefore, S_1 cannot distinguish from the transcripts if it is interacting with S_2 or Sim_{s_1} . For the subroutine RecoverEnc , the privacy for both S_1 and S_2 is captured similarly by constructing the simulator Sim_{s_2} and Sim_{s_1} for them. The simulator Sim_{s_2} , as S_1 , gets the public key pk_p and on input of $\mathcal{E}^2(\text{Enc}(c))$, Sim_{s_2} pick a random element $r \xleftarrow{\$} \mathbb{Z}_n$, and send $\mathcal{E}^2(\text{Enc}(c + r))$ to S_2 . Upon receiving $\text{Enc}(c')$ from S_2 , Sim_{s_2} then computes the $\text{Enc}(c') \cdot \text{Enc}(-r)$. S_2 cannot distinguish if it is interacting with Sim_{s_2} or S_1 since the semantic security of encryption. Similarly, Sim_{s_1} who holds the secret key sk_p , simply just follow RecoverEnc by decrypting the $\mathcal{E}^2(\text{Enc}(c + r))$ and sends back $\text{Enc}(c + r)$. During the execution of RecoverEnc , neither S_1 or S_2 can distinguish if they interact with the real server or the simulator because of the semantic secure encryption scheme. \blacksquare